Plataforma do Aplicativo JBoss Enterprise 5
Guia do [illegible] do JBoss Microcontainer

para uso com a Plataforma do Aplicativo do JBoss Enterprise
Edição 5.1.0

Mark Newton Aleš Justin

Plataforma do Aplicativo JBoss Enterprise 5 Guia do [illegible] do JBoss Microcontainer

para uso com a Plataforma do Aplicativo do JBoss Enterprise
Edição 5.1.0

Mark Newton
Red Hat
mark.newton@jboss.org

Aleš Justin
Red Hat
ajustin@redhat.com

Editado por

Misty Stanley-Jones
Red Hat
misty@redhat.com

Resumo

Este guia é direcionado para desenvolvedores do Java que desejam usar o JBoss Microcontainer para implantar ambientes modulares e personalizados do Java em seus aplicativos.

Índice

Prefácio

1. Convenções de Documentos

Este manual usa diversas convenções para destacar certas palavras e frases e chamar a atenção para informações específicas.

Em PDF e edições de texto, este manual usa tipos de letras retiradas do conjunto Liberation Fonts. O conjunto de Fontes Liberation Fonts, também é usado em formato HTML, caso o conjunto esteja instalado em seu sistema. Caso ainda não esteja, como forma alternativa, estão disponíveis tipos de letras equivalentes. Nota: O Red Hat Enterprise Linux 5 e versões mais recentes do mesmo, incluem o conjunto Liberation Fonts por padrão.

1.1. Convenções Tipográficas

São usadas quatro convenções tipográficas para realçar palavras e frases específicas. Estas convenções, e circunstâncias a que se aplicam, são as seguintes:

`Negrito Espaço Único` (Mono-spaced Bold)

Usada para realçar entradas do sistema, incluindo comandos de shell, nomes de arquivos e caminhos. São também usadas para realçar teclas Maiúsculas/Minúsculas e as combinações de teclas. Por exemplo:

> Para ver o conteúdo do arquivo **`my_next_bestselling_novel`** em sua pasta de trabalho atual, insira o comando **`cat my_next_bestselling_novel`** na janela de solicitação e pressione **`Enter`** para executar o comando.

O comando acima inclui um nome de arquivo, um comando de shell e uma tecla, todos apresentados em Negrito Espaço Único (Mono-spaced Bold) e todos distintos, graças ao conteúdo.

As combinações de tecla podem ser diferenciadas de uma tecla individual pelo sinal positivo que conecta cada parte da combinação da tecla. Por exemplo:

> Pressione **`Enter`** para executar o comando.
>
> Pressione **`Ctrl`**+**`Alt`**+**`F2`** para trocar ao terminal virtual.

A primeira sentença, destaca uma tecla específica a ser pressionada. A segunda destaca duas combinações de teclas: um conjunto de três teclas pressionadas simultaneamente.

Caso o código fonte seja discutido, serão apresentados como acima, os nomes de classe, métodos, funções, nomes de variantes e valores retornados mencionados em um parágrafo, em **`Negrito de Espaço Único (Mono-spaced Bold)`**. Por exemplo:

> Classes baseadas em arquivo, incluem **`filesystem`** para sistemas de arquivo, **`file`** para arquivos, e **`dir`** para diretórios. Cada classe possui seu conjunto próprio de permissões associadas.

Negrito Proporcional

Esta representa as palavras e frases encontradas no sistema, incluindo os nomes de aplicativos, texto de caixa de diálogo, botões rotulados, caixa de seleção e rótulos de botão de opção, títulos de menus e sub-menus. Por exemplo:

> Escolha **Sistema** → **Preferências** → **Mouse** da barra do menu principal para lançar **`Mouse Preferences`**. Na aba **`Botões`** selecione o **`Botão da esquerda do mouse`** selecione a caixa e cliquem em**`Fechar`**para mudar o botão inicial do mouse da esquerda para a direita (tornando o mouse adequado para o uso na mão esquerda).

> Selecione **Applications** → **Accessories** → **Character Map** a partir da barra de menu principal, com o objetivo de inserir um caractere especial ao arquivo **`gedit`**. Em seguida, selecione **Search** → **Find...** a partir da barra do menu **Character Map**, digite o nome do caractere no campo **`Search`** e clique em **`Next`**. O caractere pesquisado aparecerá destacado no **`Character Table`**. Clique duas vezes no caractere destacado para posicioná-lo no campo **`Text to copy`** e clique no botão **`Copy`**. Retorne ao seu documento e selecione **Edit** → **Paste** a partir da barra do menu **`gedit`**.

O texto acima inclui nomes de aplicativos, nomes de menu e itens de todo o sistema, nomes de menu específicos do aplicativo, e botões e textos encontrados na Interface Gráfica GUI, todos apresentados em Negrito Proporcional (Proportional Bold) e todos diferenciados de acordo com o contexto.

`Itálico em Negrito de Espaço Único (Mono-spaced Bold Italic)` ou ***`Itálico em Negrito Proporcional (Proportional Bold Italic)`***

Sendo o Negrito Espaço Único (Mono-spaced Bold) ou Negrito Proporcional (Proportional Bold), os itálicos extras indicam textos substituíveis ou variáveis. O Itálico denota o texto que você não inseriu literalmente ou textos exibidos que mudam dependendo das circunstâncias. Por exemplo:

> Para conectar-se à uma máquina remota usando o ssh, digite **`ssh`** ***`nome do usuário`*****`@`*****`domain.name`*** na janela de comandos. Por exemplo, considere que a máquina remota seja **`example.com`** e seu nome de usuário nesta máquina seja john, digite **`ssh john@example.com`**.

> O comando **`mount -o remount`** ***`file-system`*** remonta o sistema de arquivo nomeado. Por exemplo, para remontar o sistema de arquivo **`/home`**, o comando é **`mount -o remount /home`**.

> Para ver a versão de um pacote instalado, use o comando **`rpm -q`** ***`package`***. Ele retornará um resultado como este: ***`package-version-release`***.

Perceba as palavras em negrito e itálico acima - username, domain.name, file-system, package, version e release. Cada palavra é um espaço reservado, tanto para o texto que você insere quando emitindo um comando ou para textos exibidos pelo sistema.

Além de uso padrão para apresentar o título de um trabalho, os itálicos denotam a primeira vez que um termo novo e importante é usado. Por exemplo:

> O Publican é um sistema de publicação do *DocBook*.

1.2. Convenções de Pull-Quote

Resultado de terminal e listagem de código fonte são definidos visualmente com base no contexto.

O resultado enviado à um terminal é configurado em **`Romano de Espaço Único (Mono-spaced Roman)`** e apresentado assim:

```
books        Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

As listas de código fonte também são configuradas em **Romano de Espaço Único (Mono-spaced Roman)**, porém são apresentadas e realçadas como a seguir:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }
}
```

1.3. Notas e Avisos

E por fim, usamos três estilos visuais para chamar a atenção para informações que possam passar despercebidas.

Nota

Uma nota é uma dica ou símbolo, ou ainda uma opção alternativa para a tarefa em questão. Se você ignorar uma nota, provavelmente não resultará em más consequências, porém poderá deixar passar uma dica importante que tornará sua vida mais fácil.

Importante

Caixas importantes detalham coisas que são geralmente fáceis de passarem despercebidas: mudanças de configuração que somente se aplicam à sessão atual, ou serviços que precisam ser reiniciados antes que uma atualização seja efetuada. Se você ignorar estas caixas importantes, não perderá dados, porém isto poderá causar irritação e frustração.

Atenção

Um Aviso não deve ser ignorado. Se você ignorar avisos, muito provavelmente perderá dados.

2. Obtendo Ajuda e Fornecendo Comentários

2.1. Você precisa de ajuda?

Caso encontre dificuldades com o procedimento descrito nesta documentação, você pode encontrar

ajuda visitando o Portal do Cliente da Red Hat em http://access.redhat.com. Você poderá realizar o seguinte através do portal do cliente:

- realizar buscas ou navegar através da base de conhecimento dos artigos de suporte técnico sobre os produtos da Red Hat.
- inserir o caso de suporte nos Serviços de Suporte Global da Red Hat (GSS).
- acessar outra documentação da Red Hat.

A Red Hat acomoda um grande número de listas de correio eletrônico para discussão de software e tecnologias da Red Hat. Você pode encontrar uma lista contendo as listas públicas disponíveis em https://www.redhat.com/mailman/listinfo. Clique no nome de qualquer lista para subscrever àquela lista ou acessar o histórico das listas.

2.2. Nós precisamos do seu Comentário!

Caso você encontre um erro de tipografia ou saiba uma melhor forma de escrever este guia, nós queremos muito ouvir sua opinião. Por favor submeta um relatório em relação ao produto **JBoss Enterprise Application Platform 5** e ao componente **doc-JBoss_Microcontainer_User_Guide**. O seguinte link o levará a um pré-preenchimento deste relatório de bug para este produto: http://bugzilla.redhat.com/.

Preencha o seguinte modelo no campo **Description** do Bugzilla. Seja bastante específico na descrição do problema, uma vez que isto nos ajudará a resolver o problema com maior agilidade.

```
URL do Documento:


Nome e Número da seção:


Descrição do problema:


Sugestões de Aperfeiçoamento:


Informação adicional:
```

Por favor forneça o seu nome para que tenha o reconhecimento merecido por relatar este problema.

Parte I. Introdução ao Microcontainer - Guia Tutorial

Capítulo 1. Pré-requesitos para uso deste Guia

Você precisa instalar e configurar alguns softwares de suporte para uso das amostras deste guia, além de baixar o código para as amostras.

1.1. Instalação do Maven

As amostras usadas neste projeto requerem o **Maven** v2.2.0 ou mais avançado. Baixe o **Maven** diretamente da página principal do Apache Maven, instalando e configurando seu sistema conforme descrito no Procedimento 1.1, "Instalação do Maven".

Procedimento 1.1. Instalação do Maven

1. **Certifique-se de que o Java Developer Kit 1.6 ou superior está instalado. Isto é também solicitado para a Plataforma Enterprise.**

   Verifique de que possui o **Java** instalado em seu sistema e determine a variável do ambiente **JAVA_HOME** em seu **~/.bash_profile** para o Linux ou nas Propriedades de Sistema para o Windows. Para maiores informações sobre a configuração das variáveis de ambiente, por favor refira-se à descrição sobre este procedimento no Passo 4.

2. **Realizando o download do Maven**

   Nota

   Este passo e passos seguintes assumem que você já salvou o Maven à localização sugerida de seu sistema operacional. O Maven, como qualquer outra aplicação do Java, está apto a ser instalado em qualquer localização em seu sistema.

   Visite http://maven.apache.org/download.html.

   Clique no link do arquivo zip compilado, por exemplo **apache-maven-2.2.1-bin.zip**

   Selecione um download espelhado da lista.

   Usuários do Linux

   Salve o arquivo zip em seu diretório **home**.

   Usuários do Windows

   Salve o arquivo zip em seu diretório **C:\Documents and Settings*user_name***.

3. **Instalação do Maven**

   Usuários do Linux

   Extraia o arquivo zip para seu diretório **home**. Caso tenha selecionado o arquivo Zip no segundo passo e não tenha renomeado o diretório, o diretório extraído é nomeado **apache-maven-*version***.

   Usuários do Windows

   Extraia o arquivo zip para C:\Program Files\Apache Software Foundation. Caso tenha selecionado o arquivo zip no segundo passo e não tenha renomeado o diretório, o diretório extraído é nomeado **apache-maven-*version***.

4. **Configuração das Variáveis de Ambiente**

Usuários do Linux

Adicione as seguintes linhas ao seu **~/.bash_profile**. Certifique-se daa alteração do ***[username]*** para seu nome de usuário atual e de que o diretório Maven é o nome do diretório atual. O número de versão pode ser diferente da versão instalada abaixo.

```
export M2_HOME=/home/[username]/apache-maven-2.2.1 export M2=$M2_HOME/bin
export
PATH=$M2:$PATH
```

A versão **Maven** que você acabou de instalar será a versão padrão utilizada, pela inclusão do **M2** no início de seu caminho. Você pode configurar o caminho de sua variável de ambiente **JAVA_HOME** à localização do JDK em seu sistema.

Usuários do Windows

Adicione as variáveis do ambiente **M2_HOME**, **M2** e **JAVA_HOME**.

a. Pressione **Start**+**Pause|Break**. A caixa de diálogo das Propriedades de Sistema será exibida.
b. Clique no tab **Advanced** e clique no botão **Environment Variables**.
c. A partir do **System Variables**, selecione **Path**.
d. Clique **Edit** e acrescente dois caminhos **Maven** usando ponto e vírgula para separação de cada entrada. As aspas não são necessárias nos caminhos.
   - Adicione a variável **M2_HOME** e determine o caminho para o **C:\Program Files\Apache Software Foundation\apache-maven-2.2.1**.
   - Adicione a variável **M2** e determine o valor para **%M2_HOME%\bin**.
e. No mesmo diálogo, crie a variável do ambiente **JAVA_HOME**:
   - Adicione a variável **%JAVA_HOME%** e determine o valor para a localização de seu JDK. Por exemplo: **C:\Program Files\Java\jdk1.6.0_02**.
f. No mesmo diálogo, atualize ou crie a variável do ambiente do Caminho:
   - Adicione a variável **%M2%** para permitir que o Maven seja executado da linha de comando.
   - Adicione a variável **%JAVA_HOME%\bin** para determinar o caminho à correta instalação do Java.
g. Clique **OK** até que a caixa de diálogo do **System Properties** encerre.

5. **Implementação de alterações ao .bash_profile**

Usuários do Linux

Com o objetivo de atualizar as alterações realizadas no .bash_profile na sessão de terminal atual, origine seu .bash_profile.

```
[localhost]$ source ~/.bash_profile
```

6. **Atualização do perfil do terminal gnome**

Usuários do Linux

Atualize o perfil do terminal para garantir que as interações subsequentes do terminal gnome (ou terminal Konsole) leiam as novas variáveis do ambiente.

a. Clique **Edit** → **Profiles**
b. Selecione **Default** e clique no botão **Edit**.
c. No diálogo **Editing Profile**, clique no tab **Title and Command**.
d. Selecione a caixa de checagem **Run command as login shell**.
e. Encerre todas as caixas de diálogo no Terminal.

7. **Verifique as alterações da variável do ambiente e instale o Maven**

Usuários do Linux

Abra um terminal e execute os seguintes comandos para verificar de que as alterações foram implementadas corretamente:

- Execute o **echo $M2_HOME**, que deve retornar o resultado final.

```
[localhost]$ echo $M2_HOME /home/username/apache-maven-2.2.1
\t
```

- Execute o **echo $M2**, que deve retornar o seguinte resultado.

```
[localhost]$ echo $M2 /home/username/apache-maven-2.2.1/bin
```

- Execute o **echo $PATH** e verifique se o diretório **Maven /bin** está incluído.

```
[localhost]$ echo $PATH /home/username/apache-maven-2.2.1/bin
```

- Execute o **which mvn** que deve exibir o caminho ao **Maven** executável.

```
[localhost]$ which mvn ~/apache-maven-2.2.1/bin/mvn
```

- Execute o **mvn -version** que deve exibir a versão **Maven**, referente à versão Java, e informação do sistema operacional.

```
[localhost]$ $ mvn -version Apache Maven 2.2.1 (r801777; 2009-08-07
05:16:01+1000) Java version:
1.6.0_0 Java home: /usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0/jre Default
locale: en_US, platform encoding: UTF-8 OS
name: "Linux" version: "2.6.30.9-96.fc11.i586" arch: "i386" Family:
"unix"
```

Usuários do Windows

Para verificar de que as alterações foram implantadas corretamente, abra um terminal e execute o seguinte comando:

- Num prompt de comando, execute **mvn -version**

```
C:\> mvn -version Apache
Maven 2.2.1 (r801777; 2009-08-06 12:16:01-0700) Java version: 1.6.0_17
Java home: C:\Sun\SDK\jdk\jre Default
locale: en_US, platform encoding: Cp1252 OS name: "windows xp" version:
"5.1" arch:
"x86" Family: "windows"
```

Você configurou o **Maven** com êxito para uso com as amostras neste guia.

1.2. Configurações Especiais do Maven para as Amostras do Microcontainer

O Maven é um sistema de construção modular que puxa dependência conforme necessário. As amostras neste guia assumem que você incluiu o bloco do XML no Exemplo 1.1, “Arquivo **settings.xml** de Amostra”, em seu **~/.m2/settings.xml** (Linux) ou **C:\Documents and Settings*username*\.m2\settings.xml** (Windows). Caso o arquivo não existir, você pode criá-lo primeiramente.

Exemplo 1.1. Arquivo `settings.xml` de Amostra

```
<settings>
  <profiles>
    <profile>
      <id>jboss.repository</id>
      <activation>
        <property>
          <name>!jboss.repository.off</name>
        </property>
      </activation>
      <repositories>
        <repository>
          <id>snapshots.jboss.org</id>
          <url>http://snapshots.jboss.org/maven2</url>
          <snapshots>
            <enabled>true</enabled>
          </snapshots>
        </repository>
        <repository>
          <id>repository.jboss.org</id>
          <url>http://repository.jboss.org/maven2</url>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>repository.jboss.org</id>
          <url>http://repository.jboss.org/maven2</url>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </pluginRepository>
        <pluginRepository>
          <id>snapshots.jboss.org</id>
          <url>http://snapshots.jboss.org/maven2</url>
          <snapshots>
            <enabled>true</enabled>
          </snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile>
  </profiles>
</settings>
```

1.3. Realizando o download das Amostras

As amostras deste guia apresentam como criar um projeto maven que depende do JBoss Microcontainer, usando Maven. Você pode realizar o download das mesmas a partir do images/examples.zip. Esta localização está sujeita a alterações, mas continua inclusa para melhor conveniência.

Após você ter realizado o download do arquivo ZIP contendo as amostras, extraia-o a uma localização conveniente e verifique as amostras para familiarizar-se com sua estrutura.

Capítulo 2. Introdução ao Microcontainer

O JBoss Microcontainer é uma refatoração do JBoss JMX Microkernel para suporte direto da implantação POJO e uso autônomo fora do servidor do aplicativo JBoss.

O Microcontainer é designado para atingir as necessidades específicas dos desenvolvedores Java que desejam usar as técnicas de programação do objeto orientado para implementar rapidamente o software. Adicionado a isto, ele permite que o software seja implantado em uma grande abrangência de dispositivos, desde plataformas de computação móveis, ambientes de grade de computação em grande escala e tudo entre os dois.

2.1. Recursos

- Todos os recursos do JMX Microkernel
- Implantação POJO direta (não há necessidade para Standard/XMBean ou MBeanProxy)
- Injeção de dependência do estilo IOC Direto
- Gerenciamento aprimorado do ciclo de vida
- Controle adicional sobre dependências
- Integração AOP Transparente
- Sistema de Arquivo Virtual
- Framework de Implantação Virtual
- OSGi classloading

2.2. Definições

Este guia usa alguns termos que talvez não sejam familiares. Você poderá encontrar alguns exemplos dos mesmos na Lista de Definição do Microcontainer.

Lista de Definição do Microcontainer

JMX Microkernel

O JBoss JMX Microkernel é um ambiente Java modular. Ele difere-se dos ambientes padrões como J2EE de maneira que o desenvolvedor é apto a escolher exatamente quais componentes fazem parte do ambiente e deixar o restante.

POJO

Um *Plain Old Java Object (POJO)* é um objeto Java reutilizável e modular. O nome é usado para enfatizar que um objeto gerado é um Objeto Java comum e não é em particular um JavaBean Enterprise. O termo criado por Martin Fowler, Rebecca Parsons e Josh Mckenzie, em setembro de 2000, durante uma conversa onde eles indicavam os diversos benefícios da codificação de lógica comercial nos objetos java regulares ao invés de usar os Beans de Entidade.

Java Bean

Um Java Bean é um componente de sofware reutilizável que pode ser manipulado visualmente numa ferramenta do construtor.

Um Java Bean é um trecho de código independente. Ele não é requerido na herança de qualquer classe básica particular ou interface. Mesmo que os Java Beans sejam criados primeiramente em IDEs gráficos, eles podem também ser desenvolvidos em editores de texto

simples.

AOP

O *Aspect-Oriented Programming (AOP)* é um paradigma pelo qual funções secundárias ou de suporte são isoladas do programa principal de lógica comercial. Ele é um sub-conjunto da programação de objeto-orientado.

2.3. Isolação

O Microcontainer é parte integral da Plataforma Enterprise. Maiores informações sobre a instalação e configuração da Plataforma Enterprise podem ser encontradas no Guia de Administração e Configuração.

Capítulo 3. Serviços de Construção

Os Serviços são trechos de código que executam trabalho necessário para clientes múltiplos. Colocaremos algumas restrições adicionais na definição de um serviço para nossas finalidades. Os serviços devem possuir nomes únicos que são referenciados, ou chamados, por clientes. Os intervalos de um serviço devem ser invisíveis e sem importância aos clientes. Este é o conceito "black box" do object-oriented programming (OOP). No OOP, cada objeto é independente e nem um outro objeto precisa saber como cada um realiza seu trabalho.\n\n

Os serviços são construídos a partir de POJOS no contexto do Microcontainer. Um POJO é basicamente um serviço em sua própria função, mas não pode ser acessado por um nome único e deve ser criado pelo cliente que necessita disto.

Não é necessário implementar um POJO por classes separadas com o objetivo de fornecer uma interface bem definida, mesmo que um POJO tenha que ser criado no período de execução pelo cliente. Não há necessidade de recompilar clientes para uso do recém criado POJO, contando que os campos e métodos não sejam removidos e o acesso aos mesmos não seja restrito.

Nota

A implementação da interface é apenas necessária para permitir que um cliente escolha entre *implementações alternativas*. Caso o cliente seja compilado em referência à interface, diferentes implementações da interface podem ser fornecidas sem ter que recompilar o cliente. A interface garante que as assinaturas do método não sejam alteradas.

O restante deste guia consiste na criação de um serviço de Recursos Humanos, usando o Microcontainer para capturar e modalizar a lógica comercial do aplicativo. Após o Microcontainer ser instalado, o código de amostra pode ser encontrado em **`examples/User_Guide/gettingStarted/humanResourcesService`**.

3.1. Introdução às Amostras dos Recursos Humanos

Ao familiarizar-se com a estrutura destes arquivos na amostra, perceba que o Maven Standard Directory Layout é usado.

Os arquivos de fonte estão localizados em pacotes sob o diretório **`examples/User_Guide/gettingStarted/humanResourcesService/src/main/java/org/jboss/example/service`**, após ter extraído o arquivo ZIP. Cada uma destas classes representam um POJO simples que não implementa quaisquer interfaces especiais. A classe mais importante é a **`HRManager`**, que representa o ponto de entrada de serviço servindo todos os métodos públicos que clientes irão chamar.

Métodos fornecidos pela Classe `HRManager`

- **`addEmployee`**(Employee ***`employee`***)
- **`removeEmployee`**(Employee ***`employee`***)
- **`getEmployee`**(String ***`firstName`***, String ***`lastName`***)
- **`getEmployees`**()
- **`getSalary`**(Employee ***`employee`***)
- **`setSalary`**(Employee ***`employee`***, Integer ***`newSalary`***)
- **`isHiringFreeze`**()

- **setHiringFreeze**(boolean ***hiringFreeze***)
- **getSalaryStrategy**()
- **setSalaryStrategy**(SalaryStrategy ***strategy***)

O Serviço de Recursos Humanos é composto de uma porção de classes que mantém uma lista de funcionários e seus detalhes (endereços e salários, neste caso). É possível configurar o HRManager com o uso da interface **SalaryStrategy** de forma que as implementações de estratégias de salários estão disponíveis para definir os limites menores e maiores de salários para diferentes funções de funcionários.

3.2. Compilando o Projeto de Amostra do HRManager

Emita o comando **mvn compile** a partir do diretório **humanResourcesService/** para compilar este código de fonte. Isto cria um novo diretório chamado **target/classes** que contém as classes compiladas. E, para limpar o projeto e remover o diretório de destino emita o comando **mvn clean**.

3.3. Criação de POJOs

Antes de um POJO pode ser usado, você precisa criá-lo. Você precisa de um mecanismo de nomeação que o permite registrar uma referência à instância de POJO com um nome. Os clientes precisam deste nome para usar o POJO.

O Microcontainer fornece tal mecanismo: o *Controlador*. O Controlador permite você implantar seus serviços baseados no POJO em um ambiente do período de execução.

3.3.1. Descritores de Implantação XML

Após compilar esta classes, use um descritor de implantação XML para criar instância das mesmas. O descritor contém uma lista de beans representando instâncias individuais. Cada bean possui um nome único, de forma que pode ser chamado por clientes no período de execução. O seguinte descritor implanta uma instância do HRManager:

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:jboss:bean-deployer:2.0 bean-deployer_2_0.xsd"
xmlns="urn:jboss:bean-deployer:2.0">
  <bean name="HRService" class="org.jboss.example.service.HRManager"/>
</deployment>
```

Este XML cria uma instância da classe **HRManager** e a registra com o HRService. Este arquivo é passado a um implantador XML associado com o Microcontainer no período de execução, que executa a implantação atual e instância os beans.

3.4. Conectando POJOs

Instâncias de POJOs individuais podem apenas fornecer um comportamento relativamente simples. A potência verdadeira dos POJOs vem da conexão dos mesmos para executar as tarefas mais complexas. Como você pode ligar POJOS para escolher as implementações de estratégias de salários diferentes?

O seguinte descritor de implantação XML realiza isto:

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:jboss:bean-deployer:2.0 bean-deployer_2_0.xsd"
xmlns="urn:jboss:bean-deployer:2.0">
  <bean name="HRService" class="org.jboss.example.service.HRManager">
    <property name="salaryStrategy"><inject bean="AgeBasedSalary"/></property>
  </bean>
  <bean name="AgeBasedSalary"
class="org.jboss.example.service.util.AgeBasedSalaryStrategy"/>
</deployment>
```

Este XML cria uma instância da implementação da estratégia de salário escolhido pela inclusão de um elemento <bean> adicional. O AgeBasedSalaryStrategy é escolhido desta vez. Em seguida, o código injeta uma referência a este bean na instância do **HRManager** criado usando o bean do HRService. A injeção é possível uma vez que a classe **HRManager** contém um método **setSalaryStrategy(SalaryStrategy strategy)**. O JBoss Microcontainer chama este método na instância recém criada do HRManager e passa a referência à instância **AgeBasedSalaryStrategy**.

O descritor da implantação XML leva a mesma sequência de eventos a ocorrerem como se você estivesse escrito o seguinte código:

```
HRManager hrService = new HRManager();
AgeBasedSalaryStrategy ageBasedSalary = new AgeBasedSalaryStrategy();
hrService.setSalaryStrategy(ageBasedSalary);
```

Adicionado à injeção de execução através dos métodos setter de propriedade, o JBoss Microcontainer pode executar a injeção através dos parâmetros do construtor, caso isto seja necessário. Para maiores informações sobre este respeito, por favor consulte o capítulo 'Injeção' na Parte II 'POJO Development.' \n\t\n

3.4.1. Considerações Especiais

Embora seja possível criar instâncias de classes usando o elemento <bean> no descritor da implantação, nem sempre é a melhor maneira. Por exemplo, a criação de instâncias das classes **Employee** e **Address** é desnecessária, uma vez que o cliente as criam como resposta à entrada do usuário. Elas continuam fazendo parte do serviço, porém não são referenciadas no descritor da implantação.

Comente seu código

Você pode definir beans com um descritor de implantação contanto que cada um possua um nome único, que é usado para executar injeção conforme acima. No entanto, todos os beans não representam necessariamente serviços. Enquanto um serviço pode ser implementado usando um bean único, beans múltiplos são normalmente usados juntos. Um bean representa o ponto de entrada do serviço e contém os métodos públicos chamados pelos clientes. Neste exemplo o ponto de entrada é o bean do HRService. O descritor de implantação XML não indica se é que o bean representa um serviço ou se um bean é um ponto de entrada do serviço. É uma boa ideia usar comentários e um esquema de nomeação para delinear beans de serviço a partir de beans sem serviço.

3.5. Trabalhando com Serviços

Após criar POJOs e conectá-los para formarem serviços, você precisa configurar serviços, testá-los e empacotá-los.

3.5.1. Configuração de um Serviço

Os serviços podem ser configurados por pelo menos duas maneiras:

- Injeção de referências entre instâncias do POJO
- Injeção de valores nas propriedades do POJO

Neste exemplo, o segundo método é usado. O seguinte descritor da implantação configura a instância HRManager das seguintes maneiras:

- Um congelamento oculto é implantado.
- O **AgeBasedSalaryStrategy** implementa valores mínimo e máximo de salários.

A Injeção de referências entre instâncias de POJO é uma maneira de configurar um serviço, no entanto nós podemos injetar também valores nas propriedades do POJO. O seguinte descritor de implantação demonstra como podemos configurar a instância HRManager para ter um congelamento oculto e o AgeBasedSalaryStrategy para possuir os valores mínimo e máximo de salário:

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:jboss:bean-deployer:2.0 bean-deployer_2_0.xsd"
xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="HRService" class="org.jboss.example.service.HRManager">
    <property name="hiringFreeze">false</property>
    <property name="salaryStrategy"><inject bean="AgeBasedSalary"/></property>
  </bean>

  <bean name="AgeBasedSalary"
class="org.jboss.example.service.util.AgeBasedSalaryStrategy">
    <property name="minSalary">1000</property> <property
name="maxSalary">80000</property>
  </bean>

</deployment>
```

As classes devem possuir os métodos setter de forma que os valores sejam injetados. Por exemplo, a classe **HRManager** possui um método **setHiringFreeze(boolean hiringFreeze)** e a classe **AgeBasedSalaryStrategy** possui os métodos **setMinSalary(int minSalary)** e **setMaxSalary(int maxSalary)**.

Os valores no descritor da implantação são convertidos de sequências para tipos relevantes (boolean, int,..., etc) pelo JavaBean PropertyEditors. Muitos PropertyEditors são fornecidos por padrão pelos tipos padrões, mas você pode criar seu próprio caso ache necessário. Consulte o capítulo Propriedades na Parte II 'POJO Development' para maiores detalhes.

3.5.2. Testando o Serviço

Após ter criado seus POJOs e conectá-los para formarem serviços, você precisará testá-los. O JBoss Microcontainer permite a unidade de teste individual dos POJOs assim como os serviços, através do uso de uma classe **MicrocontainerTest**.

A classe **org.jboss.test.kernel.junit.MicrocontainerTest** herda do **junit.framework.TestCase**, configurando cada teste pelo bootstrapping JBoss Microcontainer e adicionando um BasicXMLDeployer. Depois, ela busca o classpath para um descritor de implantação

XML com o mesmo nome como classe de teste, finalizando em **`.xml`** e residindo numa estrutura de diretório representando o nome do pacote da classe. Quaisquer beans encontrados neste arquivo são implantados e podem então ser acessados usando o método convencional chamado **`getBean(String name)`**.

Amostras destes descritores podem ser encontrados no Exemplo 3.1, “Listagem do Diretório `src/test/resources` Directory”.

Exemplo 3.1. Listagem do Diretório `src/test/resources` Directory

```
├── log4j.properties
└── org
└── jboss
└── example
└── service
├── HRManagerAgeBasedTestCase.xml
├── HRManagerLocationBasedTestCase.xml
├── HRManagerTestCase.xml
└── util
├── AgeBasedSalaryTestCase.xml
└── LocationBasedSalaryTestCase.xml
```

O código de teste está localizado no diretório src/test/java:

Exemplo 3.2. Listagem do Diretório `src/test/java`

```
└── org
└── jboss
└── example
└── service
├── HRManagerAgeBasedTestCase.java
├── HRManagerLocationBasedTestCase.java
├── HRManagerTestCase.java
├── HRManagerTest.java
├── HRManagerTestSuite.java
└── util
├── AgeBasedSalaryTestCase.java
├── LocationBasedSalaryTestCase.java
└── SalaryStrategyTestSuite.java
```

A classe **`HRManagerTest`** extende o **`MicrocontainerTest`** com o objetivo de determinar um número de funcionários em uso como base para os testes. Os casos de testes individuais dividem o **`HRManagerTest`** em subclasses para executar o trabalho atual. Além disso, estão incluídas classes **`TestSuite`** que são usadas para agrupar casos de testes individuais para melhor conveniência.

Com o objetivo de rodar testes, entre o **`mvn test`** a partir do diretório **`humanResourcesService/`**. Você poderá observar que alguns resultados de log **`DEBUG`** que apresentam o JBoss Microcontainer inicializando e implantando beans do arquivo XML relevante antes de rodar cada teste. No final do teste os beans são desimplantados e o Microcontainer é encerrado.

Nota

Alguns testes, tais como **`HRManagerTestCase`**, **`AgeBasedSalaryTestCase`** e **`LocationBasedSalaryTestCase`**, testam a unidade individual dos POJOS. Por outro lado, outros testes, tais como **`HRManagerAgeBasedTestCase`** e **`HRManagerLocationBasedTestCase`** testam a unidade de serviços por completo. Em ambos os casos, os testes são rodados da mesma maneira. O uso da classe MicrocontainerTest facilita a construção e conduzem testes compreensivos de qualquer parte de seu código.
As classes **`Address`** e **`Employee`** não são testadas neste guia. Fica ao seu critério testá-las.

3.5.3. Empacotando um Serviço

Após testar seu serviço, é conveniente empacotá-lo para que demais usuários possam utilizá-lo. A maneira mais simples para realizar isto é criar um JAR contendo todas as classes. Você pode escolher em incluir o descritor de implantação caso haja uma maneira padrão sensível para configurar o serviço, mas isto é opcional.

Procedimento 3.1. Empacotando um Serviço

1. **Posicione o descritor da implantação no diretório `META-INF` (opcional)**

   Caso você decida incluir o descritor de implantação, por acordo ele deve ser nomeado **`jboss-beans.xml`** e ser posicionado num diretório **`META-INF`**. Este é o layout padrão para a Plataforma Enterprise, de forma que o implementador JAR reconhece este layout e automaticamente executa a implantação.

   O descritor da implantação não está incluso na amostra dos Recursos Humanos, uma vez que o serviço é configurado pela edição do descritor diretamente, como um arquivo separado.

2. **Gerando o JAR**

   Para gerar o JAR contendo todas as classes compiladas, entre **`mvn package`** a partir do diretório **`humanResourcesService/`**.

3. **Disponibilize o JAR para outros projetos**

   Para disponibilizar o JAR a outros projetos Maven, entre **`mvn install`** para copiá-lo ao seu repositório Maven. O layout final do JAR está apresentado no Exemplo 3.3, "Listagem dos Diretórios `org/jboss/example/service` e `META-INF`".

Exemplo 3.3. Listagem dos Diretórios `org/jboss/example/service` e `META-INF`

```
`-- org
`-- jboss
`-- example
`-- service
|-- Address.java
|-- Employee.java
|-- HRManager.java
`-- util
|-- AgeBasedSalaryStrategy.java
|-- LocationBasedSalaryStrategy.java
`-- SalaryStrategy.java
`--META-INF
`-- MANIFEST.MF
`-- maven
`-- org.jboss.micrcontainer.examples
`-- humanResourceService
```

Nota

O diretório **`META-INF/maven`** é criado automaticamente pelo Maven e não será apresentado caso você use um sistema de construção diferente.

Capítulo 4. Uso de Serviços

O capítulo anterior orientou através da criação, configuração, testes e empacotamento de um serviço. O próximo passo é criar um cliente que executará o trabalho atual usando o serviço.

O cliente nesta amostra usa um *Text User Interface (TUI)* para aceitar a entrada do usuário e resultados de saída. Isto reduz o tamanho e a complexidade do código de amostra.

Todos os arquivos necessários estão localizados no diretório **`examples/User_Guide/gettingstarted/commandLineClient`**, que segue o Maven Standard Directory Layout, conforme visto no Exemplo 4.1, “Listagem para o Diretório **`examples/User_Guide/gettingstarted/commandLineClient`**”.

Exemplo 4.1. Listagem para o Diretório `examples/User_Guide/gettingstarted/commandLineClient`

```
├── pom.xml
├── src
│  ├── main
│  │  ├── assembly
│  │  │  ├── aop.xml
│  │  │  ├── classloader.xml
│  │  │  ├── common.xml
│  │  │  └── pojo.xml
│  │  ├── config
│  │  │  ├── aop-beans.xml
│  │  │  ├── classloader-beans.xml
│  │  │  ├── pojo-beans.xml
│  │  │  └── run.sh
│  │  ├── java
│  │  │  └── org
│  │  │  └── jboss
│  │  │  └── example
│  │  │  └── client
│  │  │  ├── Client.java
│  │  │  ├── ConsoleInput.java
│  │  │  ├── EmbeddedBootstrap.java
│  │  │  └── UserInterface.java
│  │  └── resources
│  │  └── log4j.properties
│  └── test
│  ├── java
│  │  └── org
│  │  └── jboss
│  │  └── example
│  │  └── client
│  │  ├── ClientTestCase.java
│  │  ├── ClientTestSuite.java
│  │  └── MockUserInterface.java
│  └── resources
│  └── jboss-beans.xml
└── target
└── classes
└── log4j.properties
```

O cliente consiste em três classes e uma interface, localizadas no diretório

org/jboss/example/client.

O **UserInterface** descreve métodos que o cliente chama no período de execução para solicitar a entrada do usuário. O **ConsoleInput** é uma implantação do **UserInterface** que cria um TUI que o usuário usa para interação com o cliente. A vantagem deste design é que você pode facilmente criar uma implantação Swing do **UserInterface** mais tarde e substituir o TUI com o GUI. Além disso, você pode simular um processo de entrada de dados com um script. Em seguida, você pode checar o comportamento do cliente automaticamente usando os casos de testes JUnit convencionais encontrados no Exemplo 3.2, "Listagem do Diretório **src/test/java** ".

Para que a construção funcione, você deve primeiramente construir e instalar o **auditAspect.jar** do diretório **examples/User_Guide/gettingStarted/auditAspect** usando o **mvn install command**. Um número de diferentes distribuições de cliente são criadas, incluindo uma baseada no AOP que baseia-se no **auditAspect.jar** sendo disponibilizado no repositório Maven local.

Caso seu **mvn install** digitado anteriormente do diretório **examples/User_Guide/gettingStarted** e o **humanResourcesService.jar** e **auditAspect.jar** já tenham sido construídos e empacotados juntamente ao cliente, este passo não será necessário.

Para compilar o código de fonte, todos os passos no Procedimento 4.1, "Compilação do Código de Fonte" são executados quando você emitir o comando **mvn package** do diretório **commandLineClient**.

Procedimento 4.1. Compilação do Código de Fonte

1. Rode as unidades de testes.
2. Construa um JAR do cliente.
3. Monte a distribuição contendo todos os arquivos necessários.

Após compilar e empacotar o cliente, a estrutura do diretório no diretório **commandLineClient/target** inclui os subdiretórios descritos no Exemplo 4.2, "Sub-diretórios do Diretório **commandLineClient/target**".

Exemplo 4.2. Sub-diretórios do Diretório commandLineClient/target

client-pojo
: Usado para chamar o serviço sem o AOP.

client-cl
: Usado para demonstrar os recursos do classloading.

client-aop
: Adição do suporte AOP. Consulte o Capítulo 5, *Adição de Comportamento ao AOP* para maiores detalhes.

Cada sub-diretório representa uma distribuição diferente com todos os scripts de chell, JARs e descritores da implantação XML necessárias para rodar o cliente em configurações diferentes. O resto deste capítulo usa a distribuição **client-pojo** encontrada no sub-diretório **client-pojo**, que está

listado no Exemplo 4.3, “Listagem do Diretório **client-pojo**”.

Exemplo 4.3. Listagem do Diretório client-pojo

```
|-- client-1.0.0.jar
|-- jboss-beans.xml
|-- lib
| |-- concurrent-1.3.4.jar
| |-- humanResourcesService-1.0.0.jar
| |-- jboss-common-core-2.0.4.GA.jar
| |-- jboss-common-core-2.2.1.GA.jar
| |-- jboss-common-logging-log4j-2.0.4.GA.jar
| |-- jboss-common-logging-spi-2.0.4.GA.jar
| |-- jboss-container-2.0.0.Beta6.jar
| |-- jboss-dependency-2.0.0.Beta6.jar
| |-- jboss-kernel-2.0.0.Beta6.jar
| |-- jbossxb-2.0.0.CR4.jar
| |-- log4j-1.2.14.jar
| `-- xercesImpl-2.7.1.jar
`-- run.sh
```

Altere o diretório **client-pojo** e digite **./run.sh** para rodar o cliente. Consulte o Exemplo 4.4, “Tela do Menu HRManager” para maiores informações.

Exemplo 4.4. Tela do Menu HRManager

```
Menu:

d) Implanta o serviço de Recursos Humanos
u) Desimplanta o serviço de Recursos Humanos

a) Adiciona funcionário
l) Lista funcionários
r) Remove funcionário
g) Obtém o salário
s) Determina o salário
t) Ativa/Desativa o congelamento oculto

m) Menu de exibição
p) Imprime o status do serviço
q) Encerra
>
```

Para selecionar uma opção, entre a letra apresentada no lado esquerdo e pressione **RETURN**. Por exemplo, para exibir as opções do menu entre **m** seguido de **RETURN**. A inserção de mais de uma letra ou de uma opção resulta numa mensagem de erro.

Importante

O script **`run.sh`** configura o ambiente de período de rodagem pela adição de todos os JARs no diretório **`lib/`** para o classpath usando a propriedade de sistema java.ext.dirs. Ele também adiciona o diretório atual e o **`client-1.0.0.jar`** usando o aviso -cp, de forma que o descritor de implantação **`jboss-beans.xml`** pode ser encontrado no período de execução, juntamente com a classe **`org.jboss.example.client.Client`**, que é chamada no início do aplicativo.

4.1. Aplicando o Bootstrapping ao Microcontainer

Antes de usar o cliente para implementar e chamar seu serviço, observe com atenção no que acontece durante a construção:

```
public Client(final boolean useBus) throws Exception {
    this.useBus = useBus;

    ClassLoader cl = Thread.currentThread().getContextClassLoader();
    url = cl.getResource("jboss-beans.xml");

    // Start JBoss Microcontainer
    bootstrap = new EmbeddedBootstrap();
    bootstrap.run();

    kernel = bootstrap.getKernel();
    controller = kernel.getController();
    bus = kernel.getBus();
}
```

Primeiramente, um URL representando o descritor de implantação **`jboss-beans.xml`** é criado. Isto é mais tarde solicitado de forma que o implantador XML pode implantar e desimplantar beans declarados no arquivo. O método **`getResource()`** do classloader do aplicativo é usado uma vez que o arquivo **`jboss-beans.xml`** está incluído no classpath. Isto é opcional, o nome e localização do descritor de implantação não são importantes contanto que o URL seja válido e acessível.

Depois, uma instância do JBoss Microcontainer é criada, juntamente com um implantador XML. Este processo é chamado *bootstrapping* e uma classe de conveniência chamada **`BasicBootstrap`** é fornecida como parte do Microcontainer para permitir a configuração programática. Para adicionar um implantador XML, extenda o **`BasicBootstrap`** para criar uma classe e substituir o método **`bootstrap()`** protegido, conforme abaixo:

```
public class EmbeddedBootstrap extends BasicBootstrap {
    protected BasicXMLDeployer deployer;
    public EmbeddedBootstrap() throws Exception {
 super();
    }

    public void bootstrap() throws Throwable {
 super.bootstrap();
 deployer = new BasicXMLDeployer(getKernel());
 Runtime.getRuntime().addShutdownHook(new Shutdown());
    }

    public void deploy(URL url) {
 ...
 deployer.deploy(url);
 ...
     }

    public void undeploy(URL url) {
 ...
 deployer.undeploy(url);
 ...
     }

    protected class Shutdown extends Thread {
 public void run() {
     log.info("Shutting down");
     deployer.shutdown();
 }
    }
}
```

O gancho **shutdown** garante que quando o JVM sair, todos os beans são desimplantados de forma correta. Os métodos **deploy**/**undeploy** delegam ao **BasicXMLDeployer**, de forma que os beans declarados no **jboss-beans.xml** podem ser implantados e desimplantados.

Finalmente, as referências ao controlador Microcontainer e bus são restauradas, de forma que você pode pesquisar referências de bean pelo nome e acessá-las diretamente ou indiretamente conforme necessário.

4.2. Implantação do Serviço

Após criar o cliente, você pode implantar o serviço de Recursos Humanos. Isto é feito pela entrada da opção **d** a partir do TUI. A saída indica que o **BasicXMLDeployer** analisou o arquivo **jboss-beans.xml** usando o URL e instanciando os beans encontrados no mesmo.

Nota

O Microcontainer está apto a instanciar os beans uma vez que suas classes estão disponíveis no classpath de extensão dentro do arquivo **lib/humanResourcesService.jar**. Você pode ainda substituir estas classes numa estrutura de diretório destacada e adicioná-lo ao classpath do aplicativo, porém empacotá-los num JAR é basicamente mais conveniente.

O descritor da implantação é totalmente separado do arquivo **humanResourcesService.jar**. Isto

facilita a edição disto para fins de teste. O arquivo **jboss-beans.xml** no exemplo contém alguns fragmentos comentados do XML que apresentam algumas das configurações possíveis.

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="urn:jboss:bean-deployer:2.0 bean-deployer_2_0.xsd"
     xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="HRService" class="org.jboss.example.service.HRManager">
    <!-- <property name="hiringFreeze">true</property>
  <property name="salaryStrategy"><inject bean="AgeBasedSalary"/></property> -->
  </bean>

  <!-- <bean name="AgeBasedSalary"
class="org.jboss.example.service.util.AgeBasedSalaryStrategy">
       <property name="minSalary">1000</property>
       <property name="maxSalary">80000</property>
     </bean>

     <bean name="LocationBasedSalary"
    class="org.jboss.example.service.util.LocationBasedSalaryStrategy">
       <property name="minSalary">2000</property>
       <property name="maxSalary">90000</property>
       </bean> -->

</deployment>
```

Importante

Dependendo de como você acessa o serviço no período de execução, você precisará encerrar o aplicativo e reiniciá-lo novamente para reimplementar o serviço e ver suas alterações. Isto reduz a flexibilidade do aplicativo, mas resulta em um desempenho mais rápido no período de execução. Alternativamente, você pode estar apto a reimplantar o serviço enquanto o aplicativo estiver rodando. Isto aumenta a flexibilidade, mas resulta num desempenho mais lento no período de execução. Leve em consideração estas opções quando designando seus aplicativos.

4.3. Acesso Direto

Caso nenhum parâmetro seja gerado ao script **run.sh** quando o cliente for inicializado, uma referência ao **HRService** bean é pesquisada usando o controlador Microcontainer após o serviço ser implantado:

```
private HRManager manager;
...
private final static String HRSERVICE = "HRService";

...
void deploy() {
    bootstrap.deploy(url);
    if (!useBus && manager == null) {
 ControllerContext context = controller.getInstalledContext(HRSERVICE);
 if (context != null) { manager = (HRManager) context.getTarget(); }
    }
}
```

Ao invés de pesquisar imediatamente uma referência para a instância do bean, a amostra primeiro pesquisa uma referência para um **ControllerContext**, e depois obtém uma referência à instância do bean a partir do contexto usando o método **getTarget()**. O bean pode existir com o Microcontainer em qualquer um dos estados listados nos Estados do Bean com o Microcontainer.

Estados do Bean com o Microcontainer

- NOT_INSTALLED
- DESCRIBED
- INSTANTIATED
- CONFIGURED
- INSTALLED

Para controlar qual estado o bean se encontra, encapsule o mesmo em outro objeto chamando um *context* que descreve o mesmo estado atual. O nome do contexto é o mesmo do nome do bean. Uma vez que o contexto alcança o estado INSTALLED, o bean que o representa é considerado implantado.

Após criar uma referência à instância do bean representando o ponto de entrada do serviço, você pode chamar métodos para executar o trabalho:

```
@SuppressWarnings("unchecked")

    Set<Employee> listEmployees() {
    if (useBus)
 ...
    else
 return manager.getEmployees();
}
```

O cliente está acessando o serviço diretamente uma vez que ele está usando uma referência à instância do bean atual. O desempenho é bom, uma vez que cada chamada de método vai diretamente ao bean. No entanto, o que aconteceria caso você deseje configurar o serviço e reimplantá-lo enquanto o aplicativo estiver rodando?

A reconfiguração é atingida pelas alterações realizadas ao descritor de implantação XML e salvando o arquivo. Com o objetivo de reimplantar o serviço, a instância atual deve se desimplantada. Durante a desimplantação o controlador do Microcontainer libera suas referências à instância do bean, juntamente com quaisquer outros beans dependentes. Estes beans subsequentemente serão disponibilizados para a coleção de lixo, uma vez que eles não serão mais solicitados pelo aplicativo. A reimplantação do serviço cria novas instâncias de bean representando uma nova configuração. Quaisquer pesquisas subsequentes dos clientes restaurará as referências a estas novas instâncias e elas estarão aptas a acessar o serviço reconfigurado.

O problema é que a referência à instância do bean representando nosso ponto de entrada do serviço sofre o cache quando você implanta o serviço pela primeira vez. A desimplantação do serviço não possui efeito, uma vez que a instância do bean pode ser acessada usando a referência com cache e não será coletada como lixo até que o cliente a libere. Além disso, a implantação do serviço não causará outra busca uma vez que o cliente já possui uma referência com cache. Portanto, ela continuará a usar a instância de bean representando a configuração de serviço inicial.

Você pode testar este comportamento digitando **u** seguido por **RETURN** para desimplantar o serviço atual. Você deverá estar apto a acessar o serviço a partir do cliente, mesmo que ele esteja 'desimplantado'. Em seguida, realize alterações na configuração usando o arquivo **jboss-beans.xml**, salve o arquivo e implante novamente usando a opção **d**. A impressão do serviço usando a opção **p**

apresentará que o cliente continua acessando a instância inicial do serviço que foi implantado.

Atenção

Mesmo que você modifique o cliente para buscar uma nova referência toda vez em que o serviço for reimplantado, novos desenvolvedores podem distribuir cópias desta referência aos objetos por equívoco. Caso todas estas referências não forem esvaziadas durante a reimplantação, o mesmo problema de cache poderá ocorrer.

Para reimplantar o serviço reconfigurado com confiança, encerre o aplicativo por completo usando a opção 'q' e reinicie-a usando o script **run.sh**. Este é um comportamento perfeitamente aceitável para os serviços empresariais tais como Transactions, Messaging e Persistence uma vez que eles sempre estão em uso. Eles não podem ser reimplantados no período de execução e também aproveitam o alto desempenho gerado pelo uso de acesso direto. Caso seu serviço falhar se encaixar em uma dessas categorias, considere o acesso direto através do controlador Microcontainer.

4.4. Acesso Indireto

O script **run.sh** pode ser chamado com um bus de parâmetro opcional, que leva a chamadas aos Recursos Humanos para uso com o bus do Microcontainer.

Ao invés de usar uma referência direta à instância de bean obtida do controlador do Microcontainer, o novo comportamento é chamar um método no bus, passando o nome de bean, nome de método e argumentos e tipos de método. O bus usa esta informação para chamar o bean pelo cliente.

```
private final static String HRSERVICE = "HRService";

...

    @SuppressWarnings("unchecked")
 Set<Employee> listEmployees() {
 if (useBus)
     return (Set<Employee>) invoke(HRSERVICE, "getEmployees", new Object[] {},
new String[] {});
 else
     return manager.getEmployees();
    }

private Object invoke(String serviceName, String methodName, Object[] args,
String[] types) {
    Object result = null;
    try {
 result = bus.invoke(serviceName, methodName, args, types);
    } catch (Throwable t) {
 t.printStackTrace();
    }
    return result;
}
```

O bus busca pela referência à instância do bean nomeada e chama o método escolhido usando a reflexão. O cliente nunca possui uma referência à instância do bean, portanto é informado para acessar o serviço indiretamente. Uma vez que o bus não aplica o cache à referência, você pode realizar alterações à configuração do serviço e pode ser reimplantado no período de execução. As chamadas subsequentes pelo cliente serão usadas na nova referência, conforme o esperado. O cliente e o serviço

foram desacoplados.

Nota

Este comportamento pode ser testado pela implantação do serviço e pelo uso da opção **p** para imprimir o status. Desimplante o serviço usando a opção **u** e perceba que ele é inacessível. Em seguida, realize algumas alterações e implante-a novamente usando a opção **d**. Imprima o status novamente usando a opção **p**. O cliente está acessando a nova configuração do serviço.

O bus é mais lento que o acesso direto, uma vez que o bus usa a reflexão para chamar as instâncias do bean. O benefício desta abordagem é que apenas o bus possui referências às instâncias de bean. Quando um serviço for reimplantado, todas as referências existentes podem ser esvaziadas e substituídas por novas referências. Desta maneira, um serviço pode ser seguramente reimplantado no período de execução. Os Serviços que não são usados com muita frequência, ou que são específicos para certos aplicativos, são bons candidatos para o acesso indireto usando o bus Microcontainer. A redução do desempenho é balanceada pela flexibilidade que ele proporciona.

4.5. Classloading Dinâmico

Até agora você utilizou os classloaders do aplicativo e extensão para carregar todas as classes no aplicativo. A classpath do aplicativo é determinada pelo script **`run.sh`** usando o aviso -cp para inclusão do diretório atual e **`client-1.0.0.jar`**, conforme apresentado abaixo:

```
java -Djava.ext.dirs=`pwd`/lib -cp .:client-1.0.0.jar
org.jboss.example.client.Client $1
```

Os JARs no diretório **`lib`** foram adicionados ao classpath do classloader de extensão usando a propriedade do sistema java.ext.dirs ao invés da listagem do caminho completo para cada um dos JARs após o aviso -cp. Uma vez que a extensão **`classloader`** é o pai do aplicativo **`classloader`**, as classes do cliente estão aptas a encontrar todas as classes do Microcontainer e classes dos Recursos Humanos no período de execução.

Nota

A partir da versão 6 do Java e mais avançadas, você pode usar um curinga para incluir todos os JARs num diretório com o aviso -cp: **`java -cp `pwd`/lib/*:.:client-1.0.0.jar org.jboss.example.client.Client $1`**\n\t\n
Todas as classes no aplicativo serão adicionadas ao classpath do aplicativo e o classpath do classloader de extensão manterá os próprios valores padrões.

O que acontece caso você precise de um serviço adicional no período de execução? Caso o novo serviço de execução seja empacotado num arquivo JAR, ele deve ser visível ao classloader antes que suas classes possam ser carregadas. Uma vez que você já tenha determinado o classpath para o classloader do aplicativo (e classloader de extensão) na inicialização, não será necessário adicionar o URL ao JAR. A mesma situação é válida caso as classes do serviço forem contidas numa estrutura do diretório. As classes não serão encontradas no classloader do aplicativo a não ser que o diretório de nível superior esteja localizado no diretório atual (que está na classpath do aplicativo) e então as classes não serão encontradas pelo classloader do aplicativo.

Caso você deseje reimplantar um serviço existente, alterar algumas de suas classes, você precisará

encontrar um caminho alternativo às restrições de segurança que proíbem um classloader existente de recarregarem as classes.

O objetivo é criar um novo classloader que reconhece a localização de novas classes do serviço ou que carregam novas versões de classes de serviços existentes, com o objetivo de implantar os beans de serviço. O JBoss Microcontainer usa o elemento <classloader> do descritor de implantação para completar esta tarefa.

A distribuição **client-cl** contém o arquivo listado no Exemplo 4.5, "Listagem do Diretório **commandLineClient/target/client-cl**".

Exemplo 4.5. Listagem do Diretório commandLineClient/target/client-cl

```
|-- client-1.0.0.jar
|-- jboss-beans.xml
|-- lib
| |-- concurrent-1.3.4.jar
| |-- jboss-common-core-2.0.4.GA.jar
| |-- jboss-common-core-2.2.1.GA.jar
| |-- jboss-common-logging-log4j-2.0.4.GA.jar
| |-- jboss-common-logging-spi-2.0.4.GA.jar
| |-- jboss-container-2.0.0.Beta6.jar
| |-- jboss-dependency-2.0.0.Beta6.jar
| |-- jboss-kernel-2.0.0.Beta6.jar
| |-- jbossxb-2.0.0.CR4.jar
| |-- log4j-1.2.14.jar
| `-- xercesImpl-2.7.1.jar
|-- otherLib
| `-- humanResourcesService-1.0.0.jar
|`-- run.sh
```

O arquivo **humanResourcesService.jar** foi removido a um novo sub-diretório chamado **otherLib**. Ele não está mais disponível para tanto os classloaders do aplicativo ou extensão cujos classpaths são determinados no script run.sh:

```
java -Djava.ext.dirs=`pwd`/lib -cp .:client-1.0.0.jar
org.jboss.example.client.Client $1
```

Uma solução alternativa é criar um novo classloader durante a implantação do serviço, carregá-lo nas classes de serviço e criar instâncias dos beans. Consulte os conteúdos do arquivo **jboss-beans.xml**, para uma melhor ideia de como isto é realizado:

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="urn:jboss:bean-deployer:2.0 bean-deployer_2_0.xsd"
     xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="URL" class="java.net.URL">
    <constructor>

<parameter>file:/Users/newtonm/jbossmc/microcontainer/trunk/docs/examples/User_Gui
de/gettingStarted/commandLineClient/target/client-
cl.dir/otherLib/humanResourcesService-1.0.0.jar</parameter>
    </constructor>
  </bean>

  <bean name="customCL" class="java.net.URLClassLoader">
    <constructor>
      <parameter>
 <array>
   <inject bean="URL"/>
 </array>
      </parameter>
    </constructor>
  </bean>

  <bean name="HRService" class="org.jboss.example.service.HRManager">
    <classloader><inject bean="customCL"/></classloader>
    <!-- <property name="hiringFreeze">true</property>
  <property name="salaryStrategy"><inject bean="AgeBasedSalary"/></property> -->
  </bean>

  <!-- <bean name="AgeBasedSalary"
class="org.jboss.example.service.util.AgeBasedSalaryStrategy">
       <property name="minSalary">1000</property>
       <property name="maxSalary">80000</property>
       </bean>

<bean name="LocationBasedSalary"
class="org.jboss.example.service.util.LocationBasedSalaryStrategy">
<property name="minSalary">2000</property>
<property name="maxSalary">90000</property>
</bean> -->

</deployment>
```

1. Primeiro, crie uma instância **`java.net.URL`** chamada **`URL`**, usando a injeção de parâmetro no construtor para especificar o arquivo **`humanResourcesService.jar`** no sistema de arquivo local.
2. A seguir, crie uma instância do **`URLClassLoader`** pela injeção do bean URK no construtor como o único elemento na matriz.
3. Inclua um elemento <classloader> em sua definição de bean **`HRService`** e injete o bean **`customCL`**. Isto especifica que a classe **`HRManager`** precisa ser carregada pelo customCL classloader.

Você precisa decidir qual classloader usar para os outros beans na implantação. Todos os beans na implantação usam o classloader do contexto do segmento atual. Neste caso, o segmento que manuseia a implantação é o segmento principal do aplicativo que possui o classloader de contexto determinado

para o classloader do aplicativo na inicialização. Caso deseje, você pode especificar um classloader diferente da implantação usando o elemento <classloader>, conforme apresentado no Exemplo 4.6, "Especificação de um Classloader Diferente".

Exemplo 4.6. Especificação de um Classloader Diferente

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="urn:jboss:bean-deployer:2.0 bean-deployer_2_0.xsd"
     xmlns="urn:jboss:bean-deployer:2.0">

  <classloader><inject bean="customCL"/></classloader>

  <bean name="URL" class="java.net.URL">
    <constructor>

<parameter>file:/Users/newtonm/jbossmc/microcontainer/trunk/docs/examples/User_G
uide/gettingStarted/commandLineClient/target/client-
cl.dir/otherLib/humanResourcesService-1.0.0.jar</parameter>
    </constructor>
  </bean>

  <bean name="customCL" class="java.net.URLClassLoader">
    <constructor>
      <parameter>
 <array>
   <inject bean="URL"/>
 </array>
      </parameter>
    </constructor>
  </bean>

  ...

</deployment>
```

Isto seria necessário para permitir a reconfiguração do serviço descomentando os beans **AgeBasedSalary** ou **LocationBasedSalary**. Os classloaders especificados no nível de bean substituem o classloader no nível de implantação. Para substituir o classloader de uma só vez e usar o classloader padrão para um bean, use o valor <null/>, como segue abaixo:

```
<bean name="HRService" class="org.jboss.example.service.HRManager">
  <classloader><null/></classloader>
</bean>
```

4.5.1. Problemas com os Classloaders criados com os Descritores de Implantação

Caso você crie um novo classloader para seu serviço usando o descritor de implantação, você talvez não esteja apto a acessar as classes carregadas pelo mesmo no classloader do aplicativo. Na amostra do HRManager, o cliente não está mais apto a aplicar o cache numa referência direta à instância do bean quando usando o controlador do Microcontainer.

Para ver este comportamento, inicie o cliente usando o comando **run.sh** e tente implantar o serviço.

Uma exceção **`java.lang.NoClassDefFoundError`** será lançada e o aplicativo será encerrado.

Neste cenário, você deverá usar o bus para acessar o serviço indiretamente e fornecer acesso a quaisquer classes compartilhadas pelo cliente no classpath do aplicativo. Neste exemplo, as classes afetadas são **`Address`**, **`Employee`** e **`SalaryStrategy`**.

Capítulo 5. Adição de Comportamento ao AOP

O Object Oriented Programming (OOP) - Programação Orientada ao Objeto - contém diversas para técnicas de software incluindo encapsulamento, herança e polimorfismo. No entanto, ele não soluciona o problema de lógica de endereçamento, da qual é normalmente repetida em diferentes classes. Alguns exemplos incluem logging, segurança e lógica transacional, sendo isto codificada em cada classe. Este tipo de lógica é chamada de *cross-cutting concern*.

O *Aspect Oriented Programming (AOP)* - Programação Orientada ao Aspecto - trabalha para permitir que os cross-cutting concerns sejam aplicados as classes após terem sido compilados. Isto mantém livre o código de fonte da lógica que não é central ao propósito principal da classe e dinamiza a manutenção. Normalmente, se uma classe implementa uma interface, cada método chama uma instância da primeira classe através do proxy . Este proxy implementa a mesma interface, adicionando o comportamento requerido. Por outro lado, caso uma interface não for usada, o código bite do java da classe compilada é modificado: os métodos originais são renomeados e substituídos pelos métodos que implementam um cross-cutting logic. Depois, estes novos métodos podem chamar os métodos originais após o cross-cutting logic para criar sub-classes da classe original que substitui seus métodos. Os métodos substituídos executam o cross-cutting logic antes de chamar os métodos correspondentes da super classe.

O **JBoss AOP** é um framework para o AOP. Você pode criar cross-cutting concerns usando classes e métodos java convencionais. Na terminologia AOP, cada questão é representada por um *aspect* que você implementa usando um POJO simples. O comportamento é fornecido por métodos com o aspecto chamado *advices*. Estes avisos seguem certas regras para seus parâmetros além de retornar tipos e quaisquer exceções lançadas pelos mesmos. Você pode usar com este framework as noções do objeto orientado tais como herança, encapsulamento e composição para facilitar a manutenção dos cross-cutting concerns. Os aspectos são aplicados ao código usando uma linguagem de expressão que o permite especificar quais construtores, métodos e ainda campos a destinar. Você pode alterar rapidamente este comportamento das classes múltiplas pela edição do arquivo de configuração.

Este capítulo contém exemplos que demonstram como usar o JBoss AOP juntamente como o Microcontainer para criar e aplicar um aspecto de auditoria ao Serviço de Recursos Humanos. O código de auditoria pode ser substituído pela classe **HRManager**, no entanto isto aplicaria o clutter na classe com o código que não é relevante a seu propósito central, expandindo-o e dificultando sua operação. Este design do aspect fornece modularidade, facilitando a auditoria em outra classe futuramente, caso o escopo do projeto mudar.

O AOP pode ser usado também para aplicar um comportamento adicional durante a fase de implantação. Esta amostra criará e efetuará o bind num proxy para uma instância de bean no serviço JNDI básico, permitindo que seja acessado usando uma busca JNDI ao invés de um controlador Microcontainer.

5.1. Criando um Aspect

O diretório **examples/User_Guide/gettingStarted/auditAspect** contém todos os arquivos necessários para criar o aspect.

- **pom.xml**
- **src/main/java/org/jboss/example/aspect/AuditAspect.java**

Exemplo 5.1. POJO de Amostra

```
public class AuditAspect {

    private String logDir;
    private BufferedWriter out;

    public AuditAspect() {
        logDir = System.getProperty("user.dir") + "/log";

        File directory = new File(logDir);
        if (!directory.exists()) {
            directory.mkdir();
        }
    }

    public Object audit(ConstructorInvocation inv) throws Throwable {
        SimpleDateFormat formatter = new SimpleDateFormat("ddMMyyyy-kkmmss");
        Calendar now = Calendar.getInstance();
        String filename = "auditLog-" + formatter.format(now.getTime());

        File auditLog = new File(logDir + "/" + filename);
        auditLog.createNewFile();
        out = new BufferedWriter(new FileWriter(auditLog));
        return inv.invokeNext();
    }

    public Object audit(MethodInvocation inv) throws Throwable {
        String name = inv.getMethod().getName();
        Object[] args = inv.getArguments();
        Object retVal = inv.invokeNext();

        StringBuffer buffer = new StringBuffer();
        for (int i=0; i < args.length; i++) {
            if (i > 0) {
                buffer.append(", ");
            }
            buffer.append(args[i].toString());
        }

        if (out != null) {
            out.write("Method: " + name);
            if (buffer.length() > 0) {
                out.write(" Args: " + buffer.toString());
            }
            if (retVal != null) {
  out.write(" Return: " + retVal.toString());
            }
            out.write("\n");
            out.flush();
        }
        return retVal;
    }
}
```

Procedimento 5.1. Criação do POJO

1. O construtor checa a presença de um diretório no diretório **`log`** de trabalho atual e cria um novo,

caso não o tenha encontrado.

2. Em seguida, um advice é definido. Este advice é chamado a todo instante em que o construtor da classe de destinação é chamado. Isto cria um novo arquivo log com o diretório **`log`** para gravar as chamadas de método realizadas em instâncias diferentes da classe de destinação em arquivos separados.
3. Finalmente, outro advice é definido. Este advice é válido para cada chamada de método realizada na classe de destinação. O nome do método e argumentos são armazenados juntamente com o valor de retorno. Esta informação é usada para construir uma gravação de auditoria e gravá-la no arquivo do log atual. Cada advice chama **`inv.invokeNext()`**, que e agrupa os advices caso mais de um cross-cutting seja aplicado ou para chamar o método/construtor de destinação.

Nota

Cada advice é implementado usando um método que usa o objeto de invocação como um parâmetro, lança Throwable e retorna um Object. No período de design, você não precisa saber quais construtores ou métodos estes advices serão aplicados, portanto certifique-se de que os tipos são os mais genéricos possíveis.

Para compilar a classe e criar um arquivo **`auditAspect.jar`** que pode ser usado por outros exemplos, digite **`mvn install`** no diretório **`auditAspect`**.

5.2. Configuração do Microcontainer para o AOP

Antes de aplicar o aspect de auditoria a um Serviço HR, um número de JARs devem ser adicionados à extensão de classpath. Eles são os sub-diretórios **`lib`** da distribuição **`client-aop`** localizado na distribuição **`client-aop`** do diretório **`examples/User_Guide/gettingStarted/commandLineClient/target/client-aop.dir`**:

Exemplo 5.2. Listagem do examples/User_Guide/gettingStarted/commandLineClient/target/client-aop.dir `Directory`

```
|-- client-1.0.0.jar
|-- jboss-beans.xml
|-- lib
|-- auditAspect-1.0.0.jar
|-- concurrent-1.3.4.jar
|-- humanResourcesService-1.0.0.jar
|-- javassist-3.6.0.GA.jar
|-- jboss-aop-2.0.0.beta1.jar
|-- jboss-aop-mc-int-2.0.0.Beta6.jar
|-- jboss-common-core-2.0.4.GA.jar
|-- jboss-common-core-2.2.1.GA.jar
|-- jboss-common-logging-log4j-2.0.4.GA.jar
|-- jboss-common-logging-spi-2.0.4.GA.jar
|-- jboss-container-2.0.0.Beta6.jar
|-- jboss-dependency-2.0.0.Beta6.jar
|-- jboss-kernel-2.0.0.Beta6.jar
|-- jbossxb-2.0.0.CR4.jar
|-- log4j-1.2.14.jar
|-- trove-2.1.1.jar
`-- xercesImpl-2.7.1.jar
|-- log
`-- auditLog-18062010-122537
```

Primeiramente, o **`lib/auditAspect-1.0.0.jar`** é solicitado para criar uma instância de um aspect no período de execução, com o objetivo de executar a lógica. Em seguida, o arquivo jar para o JBoss AOP (jboss-aop.jar), juntamente com suas dependências javassist e trove adiciona a funcionalidade AOP. Finalmente, o jboss-aop-mc-int jar é requerido uma vez que ele contém uma definição do esquema XML que o permite definir aspects dentro do descritor de implantação. Ele contém também o código de integração para criar dependências entre os beans normais e os beans dos aspects com o Microcontainer, permitindo adicionar o comportamento durante as fases de implantação e desimplantação.

Uma vez que você está usando Maven2 para montar a distribuição client-aop, você deverá adicionar estes arquivos JAR pela declaração das dependências apropriadas em seu arquivo **`pom.xml`** e criação de um descritor assembly. Um trecho **`pom.xml`** de amostra é demonstrado no Exemplo 5.3, “Amostra **`pom.xml`** Excerpt para o AOP”. O procedimento será diferente para executar sua construção usando Ant.

Exemplo 5.3. Amostra `pom.xml` Excerpt para o AOP

```
<dependency>
  <groupId>org.jboss.microcontainer.examples</groupId>
  <artifactId>jboss-oap</artifactId>
  <version>2.0.0</version>
</dependency>
<dependency>
  <groupId>org.jboss.microcontainer.examples</groupId>
  <artifactId>javassist</artifactId>
  <version>3.6.0.GA</version>
</dependency>
<dependency>
  <groupId>org.jboss.microcontainer.examples</groupId>
  <artifactId>trove</artifactId>
  <version>2.1.1</version>
</dependency>
<dependency>
  <groupId>org.jboss.microcontainer.examples</groupId>
  <artifactId>jboss-aop-mc-int</artifactId>
  <version>2.0.0.Beta6</version>
</dependency>
```

5.3. Aplicação de um Aspect

Você pode configurar o **`jboss-beans.xml`** para aplicar o aspect de auditoria, uma vez que você possui uma distribuição válida contendo tudo o que você possui. Ele está no **`examples/User_Guide/gettingStarted/commandLineClient/target/client-aop.dir`**.

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="urn:jboss:bean-deployer:2.0 bean-deployer_2_0.xsd"
            xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="AspectManager" class="org.jboss.aop.AspectManager">
    <constructor factoryClass="org.jboss.aop.AspectManager"
   factoryMethod="instance"/>
  </bean>

  <aop:aspect xmlns:aop="urn:jboss:aop-beans:1.0"
       name="AuditAspect" class="org.jboss.example.aspect.AuditAspect"
       method="audit" pointcut="execution(public
org.jboss.example.service.HRManager->new(..)) OR
execution(public * org.jboss.example.service.HRManager->*(..))">
  </aop:aspect>

  ...

</deployment>
```

Procedimento 5.2. Explicação de um Código para Aplicação de um Aspect

1. Antes de você aplicar seu aspect a qualquer uma das classes, você precisa criar uma instância do **`org.jboss.aop.AspectManager`** usando um elemento <bean>. Um método de fábrica é

usado aqui ao invés de chamar um construtor convencional, sendo que apenas uma instância do AspectManager no JVM é necessária no período de execução.

2. Em seguida, uma instância de nosso aspect chamado AuditAspect é criada usando o elemento <aop:aspect>. Isto é parecido com o elemento <bean> uma vez que os atributos name e class são usados da mesma maneira. No entanto, este processo possui também os atributos method e pointcut que você pode usar para aplicar ou bind um advice com o aspect de auditoria para todos os construtores de métodos da classe **HRManager**. Apenas o método **audit** precisa ser especificado, uma vez que ele é sobrecarregado com a classe **AuditAspect** de parâmetros diferentes. O JBoss AOP sabe o que selecionar no período de execução, dependendo se um construtor ou uma invocação de método são realizados.

Esta configuração adicional é tudo o que é necessário para aplicar o aspect de auditoria no período de execução, adicionando o comportamento de auditoria ao serviço **Human Resources**. Você pode testar este procedimento rodando o cliente usando o script **run.sh**. Um diretório **log** é criado na inicialização juntamente com o diretório **lib** quando o **AuditAspect** bean é criado pelo Microcontainer. Cada implementação do serviço de Recursos Humanos leva um novo arquivo log a aparecer com o diretório **log**. O arquivo log contém uma gravação de quaisquer chamadas realizadas do cliente ao serviço. Isto é similar ao **auditLog-28112007-163902** e possui um resultado parecido com Exemplo 5.4, “Amostra do Resultado Log AOP”.

Exemplo 5.4. Amostra do Resultado Log AOP

```
Method: getEmployees Return: []
Method: addEmployee Args: (Santa Claus, 1 Reindeer Avenue, Lapland City -
25/12/1860) Return: true
Method: getSalary Args: (Santa Claus, null - Birth date unknown) Return: 10000
Method: getEmployees Return: [(Santa Claus, 1 Reindeer Avenue, Lapland City -
25/12/1860)]
Method: isHiringFreeze Return: false
Method: getEmployees Return: [(Santa Claus, 1 Reindeer Avenue, Lapland City -
25/12/1860)]
Method: getSalaryStrategy
```

Para remover o comportamento de auditoria, comentar os fragmentos relevantes do XML no descritor de implantação e reiniciar o aplicativo.

Atenção

A ordem de implantação faz diferença. Cada aspect deve ser declarado antes dos bean a ser aplicado, de forma que o Microcontainer os implementa naquela ordem. Isto é devido ao Microcontainer precisar alterar o código de byte da classe de bean normal para adicionar o cross-cutting logic, antes de criar uma instância e armazenar uma referência a mesma no controlador. Isto não será possível caso uma instância de bean normal já tenha sido criada.

5.4. Retornos de Chamada do Ciclo de Vida

Além de aplicar os aspects aos beans que instanciamos usando o Microcontainer, nós podemos adicionar também o comportamento do processo de implantação e desimplantação. Conforme mencionado na Seção 4.3, “Acesso Direto”, um bean passa por diferentes estados quando implantado, entre eles:

NOT_INSTALLED
: o descritor de implantação contendo o bean é analisado, juntamente com quaisquer anotações do próprio bean.

DESCRIBED
: quaisquer dependências criadas pelo AOP foram adicionadas ao bean e anotações personalizadas são processadas.

INSTANTIATED
: uma instância de bean é criada

CONFIGURED
: propriedades foram injetadas no bean, juntamente com quaisquer referências a outros beans.

CREATE
: o método **`create`**, caso definido no bean, é chamado.

START
: o método **`start`**, caso definido no bean, é chamado.

INSTALLED
: quaisquer ações de instalação personalizada que foram definidas no descritor de implantação foram executadas e o bean está pronto para acesso.

Importante

Os estados **`CREATE`** e **`START`** são inclusos para propósitos de legacia. Isto permite serviços que foram implementados como MBeans em versões anteriores da Plataforma Enterprise para funcionarem corretamente quando implantados como beans na Plataforma Enterprise 5.1. Caso você defina quaisquer métodos criar/iniciar em seu bean, este processo passará direto através destes estados.

Estes estados representam o ciclo de vida do bean. Você pode definir um número de retornos de chamada a qualquer instante usando um conjunto adicional de elementos <aop>:

<aop:lifecycle-describe>
: aplicado quando entrando/deixando o estado DESCRIBED

<aop:lifecycle-instantiate>
: aplicado quando entrando/deixando o estado INSTANTIATED

<aop:lifecycle-configure>
: aplicado quando entrando/deixando o estado CONFIGURED

<aop:lifecycle-create>
: aplicado quando entrando/deixando o estado CREATE

<aop:lifecycle-start>
: aplicado quando entrando/deixando o estado START

<aop:lifecycle-install>
: aplicado quando entrando/deixando o estado INSTALLED

Assim como os elementos <bean> e <aop:aspect>, os elementos <aop:lifecycle-> contém os atributos name e class. O Microcontainer usa estes atributos para criar uma instância da classe **`callback`**, nomeando-a de forma que pode ser usada como beans entrando ou deixando o estado relevante durante a implantação ou desimplantação. Você pode especificar quais beans são afetados pelo retorno de chamada usando estes atributos de classe, conforme apresentado no Exemplo 5.5, “Usando o Atributo classes”.

Exemplo 5.5. Usando o Atributo classes

```
<aop:lifecycle-install xmlns:aop="urn:jboss:aop-beans:1.0"

name="InstallAdvice"

class="org.jboss.test.microcontainer.support.LifecycleCallback"

classes="@org.jboss.test.microcontainer.support.Install">

</aop:lifecycle-install>
```

Este código especifica qual lógica adicional da classe **`lifecycleCallback`** é aplicada a quaisquer classes de bean que são anotadas com ***`@org.jboss.test.microcontainer.support.Install`***, antes de entrar e após deixar o estado **`INSTALLED`**.

Com o objetivo da classe de retorno de chamada funcionar, ela deve conter os métodos **`install`** e **`uninstall`** que levam ***`ControllerContext`*** como parâmetro, conforme apresentado no Exemplo 5.6, “Métodos de Instalação e Desinstalação”.

Exemplo 5.6. Métodos de Instalação e Desinstalação

```
import org.jboss.dependency.spi.ControllerContext;

public class LifecycleCallback {

    public void install(ControllerContext ctx) {
        System.out.println("Bean " + ctx.getName() + " is being installed";
    }
    public void uninstall(ControllerContext ctx) {
     System.out.println("Bean " + ctx.getName() + " is being uninstalled";
    }
}
```

O método **`install`** é chamado durante a implantação do bean e o método **`uninstall`** durante a própria desimplementação.

Nota

Mesmo que o comportamento tenha sido adicionado ao processo de implantação e desimplantação usando retornos de chamada, o AOP não é usado aqui. A funcionalidade *pointcut expression* do JBoss AOP é usada para determinar quais classes de bean os comportamentos são aplicados.

5.5. Adição das Pesquisas de Serviço através do JNDI

Até agora você usou o Microcontainer para referências de pesquisa para instâncias de bean que representam serviços. Isto não é recomendado uma vez que requer uma referência ao Microcontainer kernel antes do controlador ser acessado. Este processo está descrito no Exemplo 5.7, “Pesquisando Referências para Beans”.

Exemplo 5.7. Pesquisando Referências para Beans

```
private HRManager manager;
private EmbeddedBootstrap bootstrap;
private Kernel kernel;
private KernelController controller;
private final static String HRSERVICE = "HRService";

...

// Start JBoss Microcontainer
bootstrap = new EmbeddedBootstrap();
bootstrap.run();

kernel = bootstrap.getKernel();
controller = kernel.getController();

...

 ControllerContext context = controller.getInstalledContext(HRSERVICE);
if (context != null) { manager = (HRManager) context.getTarget(); }
```

A distribuição das referências de kernel para cada cliente que pesquisa um serviço é um risco de segurança, pois fornece acesso à configuração Microcontainer. Para uma melhor segurança, aplique o ServiceLocator padrão e use a classe para realizar pesquisas para os clientes. Uma opção ainda melhor é passar as referências do bean, justamente com seus nomes, ao ServiceLocator no período de implantação, usando o ciclo de vida do retorno de chamada. Neste cenário, o ServiceLocator pode observá-los sem conhecimento do Microcontainer. A desimplantação removerá subsequentemente as referências do bean a partir do ServiceLocator para prevenir pesquisas futuras.

Não seria difícil escrever sua própria implantação do ServiceLocator. A integração de uma atual tal como JBoss Naming Service (JBoss NS) é ainda mais rápida e possui um benefício adicional de compilar a especificação Java Naming e Directory Interface (JNDI). O Java JNDI ativa clientes para acessar diferentes e possivelmente múltiplos serviços de nomeação usando um API comum.

Procedimento 5.3. Escrevendo sua própria Implantação ServiceLocator

1. Primeiramente, crie uma instância do JBoss NS usando o Microcontainer.
2. Em seguida, adicione o retorno de chamada do ciclo de vida para binding e unbinding as referências de bean durante a implantação e desimplantação.
3. Marque as classes do bean que deseja realizar o bind nas referências, usando anotações.
4. Agora, você pode localizar os beans no período de execução usando a expressão abreviada pointcut conforme descrito anteriormente.

Parte II. Conceitos Avançados do Microcontainer

Esta seção cobre conceitos avançados e apresenta alguns recursos interessantes do Microcontainer. As amostras de código são consideradas amostras incompletas no resto do guia e é responsabilidade do programador extrapolar e estendê-las conforme necessário.

Capítulo 6. Modelos de Componentes

O JBoss Microcontainer trabalha com diversos modelos de componente POJO populares. Os componentes são programas de software re-utilizáveis que você pode desenvolver e agrupar com facilidade para criar aplicativos sofisticados. A integração efetiva com estes modelos de componente era um objetivo chave para o Microcontainer. Alguns modelos de componente que podem ser usados com o Microcontainer são JMX, Spring e Guice.

6.1. Interações permitidas com os Modelos de Componente

Antes de discutir a interação com alguns dos modelos de componente popular, é importante entender quais tipos de interações são permitidas. Os JMX MBeans são um exemplo de um modelo do componente. As interações dos mesmo são execuções das operações do MBean, referenciação dos atributos, configuração dos atributos e declaração explícita das dependências entre MBeans nomeados.

Os comportamentos e interações padrões no Microcontainer são o que você normalmente obtém de qualquer outro container *Inversion of Control (IoC)* e são parecidos à funcionalidade fornecida pelo MBeans, incluindo as invocações do método para operações, setters/getters para atributos e dependências explícitas.

6.2. Bean sem dependências

O Exemplo 6.1, “Descritor de Implantação para o POJO Simples” apresenta um descritor de implantação para um POJO simples sem dependências. Este é o ponto de partida para a integração do Microcontainer com o Spring ou Guice.

Exemplo 6.1. Descritor de Implantação para o POJO Simples

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="PlainPojo" class="org.jboss.demos.models.plain.Pojo"/>

  <beanfactory name="PojoFactory" class="org.jboss.demos.models.plain.Pojo">
    <property
name="factoryClass">org.jboss.demos.models.plain.PojoFactory</property>
  </beanfactory>

</deployment>
```

6.3. Usando o Microcontainer com o Spring

Exemplo 6.2. Descritor com o Suporte do Spring

```
<beans xmlns="urn:jboss:spring-beans:2.0">

  <!-- Adding @Spring annotation handler -->
  <bean id="SpringAnnotationPlugin"
class="org.jboss.spring.annotations.SpringBeanAnnotationPlugin" />

  <bean id="SpringPojo" class="org.jboss.demos.models.spring.Pojo"/>

</beans>
```

Este namespace do arquivo é diferente do arquivo do bean Microcontainer simples. O namespace **urn:jboss:spring-beans:2.0** aponta para sua versão da porta do esquema Spring, que descreve seu estilo de Spring do bean. O Microcontainer implanta os beans ao invés da noção da fábrica bean do Spring.

Exemplo 6.3. Uso do Spring com o Microcontainer

```
public class Pojo extends AbstractPojo implements BeanNameAware {
    private String beanName;

    public void setBeanName(String name)
    {
 beanName = name;
    }

    public String getBeanName()
    {
 return beanName;
    }

    public void start()
    {
 if ("SpringPojo".equals(getBeanName()) == false)
     throw new IllegalArgumentException("Name doesn't match: " + getBeanName());
    }
}
```

Mesmo que o SpringPojo bean possua dependência na biblioteca do Spring causada pela implantação da interface BeanNameAware, seu único propósito é expor e similar alguns do comportamento do retorno de chamada do Spring.

6.4. Uso do Guice com o Microcontainer

O objetivo do Guice é um tipo de combinação. Os Guice beans são gerados e configurados usando Módulos.

Exemplo 6.4. Descritor de implantação para Integração do Guice no Microcontainer

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="GuicePlugin"
class="org.jboss.guice.spi.GuiceKernelRegistryEntryPlugin">
    <constructor>
      <parameter>
 <array elementClass="com.google.inject.Module">
   <bean class="org.jboss.demos.models.guice.PojoModule"/>
 </array>
      </parameter>
    </constructor>
  </bean>

</deployment>
```

As duas partes importantes para assistir este arquivo são **`PojoModule`** e **`GuiceKernelRegistryEntryPlugin`**. O **`PojoModule`** configura seus beans conforme o Exemplo 6.5, "Configuração dos Beans para Guice". O **`GuiceKernelRegistryEntryPlugin`** fornece integração com o Microcontainer, conforme apresentado no Exemplo 6.6, "Integração do Guice com o Microcontainer".

Exemplo 6.5. Configuração dos Beans para Guice

```
public class PojoModule extends AbstractModule {
    private Controller controller;

    @Constructor
 public PojoModule(@Inject(
      bean = KernelConstants.KERNEL_CONTROLLER_NAME)
     Controller controller)
    {
 this.controller = controller;
    }

    protected void configure()
    {
 bind(Controller.class).toInstance(controller);
 bind(IPojo.class).to(Pojo.class).in(Scopes.SINGLETON);
 bind(IPojo.class).annotatedWith(FromMC.class).
     toProvider(GuiceIntegration.fromMicrocontainer(IPojo.class, "PlainPojo"));
    }
}
```

Exemplo 6.6. Integração do Guice com o Microcontainer

```
public class GuiceKernelRegistryEntryPlugin implements KernelRegistryPlugin {
    private Injector injector;

    public GuiceKernelRegistryEntryPlugin(Module... modules)
    {
 injector = Guice.createInjector(modules);
    }

    public void destroy()
    {
 injector = null;
    }

    public KernelRegistryEntry getEntry(Object name)
    {
 KernelRegistryEntry entry = null;
 try
     {
  if (name instanceof Class<?>)
      {
   Class<?> clazz = (Class<?>)name;
   entry = new AbstractKernelRegistryEntry(name, injector.getInstance(clazz));
      }
  else if (name instanceof Key)
      {
   Key<?> key = (Key<?>)name;
   entry = new AbstractKernelRegistryEntry(name, injector.getInstance(key));
      }
     }
 catch (Exception ignored)
     {
     }
 return entry;
    }
}
```

Nota

Um **`Injector`** é criado a partir de uma classe **`Modules`** e pesquisa por beans de mesma combinação. Consulte a Seção 6.5, “MBeans de Legacia e Mistura de Modelos de Componentes Diferentes” para maiores informações a respeito da declaração e uso dos MBeans de legacia.

6.5. MBeans de Legacia e Mistura de Modelos de Componentes Diferentes

A maneira mais simples de misturar modelos de conteúdos diferentes pode ser encontrado no Exemplo 6.7, “Injetando o POJO em um MBean”.

Exemplo 6.7. Injetando o POJO em um MBean

```
<server>

  <mbean code="org.jboss.demos.models.mbeans.Pojo"
name="jboss.demos:service=pojo">
    <attribute name="OtherPojo"><inject bean="PlainPojo"/></attribute>
  </mbean>

</server>
```

Para implantar a implantação MBeans através do Microcontainer, você deverá gravar um manuseador por completo para o modelo do componente. Consulte **`system-jmx-beans.xml`** para maiores detalhes. O código deste arquivo mora no código de fonte do Servidor do Aplicativo JBoss: sub-projeto *system-jmx*.

6.6. Exposição de POJOs como MBeans

Exemplo 6.8. Exposição de um POJO existente como um MBean

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="AnnotatedJMXPojo" class="org.jboss.demos.models.jmx.AtJmxPojo"/>

  <bean name="XmlJMXPojo" class="org.jboss.demos.models.mbeans.Pojo">

<annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX(exposedInterface=org.jb
oss.demos.models.mbeans.PojoMBean.class, registerDirectly=true)</annotation>
  </bean>

  <bean name="ExposedPojo" class="org.jboss.demos.models.jmx.Pojo"/>

  <bean name="AnnotatedExposePojo"
class="org.jboss.demos.models.jmx.ExposePojo">
    <constructor>
      <parameter><inject bean="ExposedPojo"/></parameter>
    </constructor>
  </bean>

</deployment>
```

O descritor expõe um POJO existente como um MBean e o registra num servidor MBean.

Para expor o POJO como um MBean, finalize-o com uma anotação @JMX assumindo que você importou o **`org.jboss.aop.microcontainer.aspects.jmx.JMX`**. O Bean pode tanto expor isto diretamente ou em sua própria propriedade.

Exemplo 6.9. Exposição de um POJO como um MBean usando s própria Propriedade

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="XMLLoginConfig"
class="org.jboss.demos.models.old.XMLLoginConfig"/>

  <bean name="SecurityConfig" class="org.jboss.demos.models.old.SecurityConfig">
    <property name="defaultLoginConfig"><inject
bean="XMLLoginConfig"/></property>
  </bean>

  <bean name="SecurityChecker" class="org.jboss.demos.models.old.Checker">
    <property name="loginConfig"><inject
bean="jboss.security:service=XMLLoginConfig"/></property>
    <property name="securityConfig"><inject
bean="jboss.security:service=SecurityConfig"/></property>
  </bean>

</deployment>
```

Você pode usar qualquer um dos tipos de pesquisa de injeção, tanto pela pesquisa de um POJO simples ou obtendo um MBean de um servidor MBean. Uma das opções da injeção é usar um tipo de injeção, às vezes chamada de *autowiring* conforme descrito no Exemplo 6.10, “Autowiring”.

Exemplo 6.10. Autowiring

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="FromGuice" class="org.jboss.demos.models.plain.FromGuice">
    <constructor><parameter><inject bean="PlainPojo"/></parameter></constructor>
    <property name="guicePojo"><inject/></property>
  </bean>

  <bean name="AllPojos" class="org.jboss.demos.models.plain.AllPojos">
    <property name="directMBean"><inject
bean="jboss.demos:service=pojo"/></property>
    <property name="exposedMBean"><inject
bean="jboss.demos:service=ExposedPojo"/></property>
    <property name="exposedMBean"><inject
bean="jboss.demos:service=ExposedPojo"/></property>
  </bean>

</deployment>
```

O *FromGuice* injeta o Guice bean através da combinação, onde o **`PlainPojo`** é injetado com uma injeção de nome comum. A partir de agora, você pode testar se o Guice binding funciona como o esperado, conforme apresentado no Exemplo 6.11, “Testando a Funcionalidade do Guice ”.

Exemplo 6.11. Testando a Funcionalidade do Guice

```
public class FromGuice {
    private IPojo plainPojo;
    private org.jboss.demos.models.guice.Pojo guicePojo;

    public FromGuice(IPojo plainPojo)
    {
 this.plainPojo = plainPojo;
    }

    public void setGuicePojo(org.jboss.demos.models.guice.Pojo guicePojo)
    {
 this.guicePojo = guicePojo;
    }

    public void start()
    {
 f (plainPojo != guicePojo.getMcPojo())
     throw new IllegalArgumentException("Pojos are not the same: " + plainPojo +
"!=" + guicePojo.getMcPojo());
    }
}
```

O Exemplo 6.11, “Testando a Funcionalidade do Guice ” apenas fornece um modelo de componente. O alias é trivial, mas um recurso necessário. Ele deve ser introduzido como um modelo de componente dentro do Microcontainer, com o objetivo de implantá-lo como uma dependência verdadeira. Os detalhes da implantação podem ser encontrados no Exemplo 6.12, “Código de Fonte AbstractController ”.

Exemplo 6.12. Código de Fonte AbstractController

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">
  <alias name="SpringPojo">springPojo</alias>
</deployment>
```

Este descritor mapeia o nome do SpringPojo para o springPojo alias. O benefício dos aliases, como modelos de componente verdadeiros, é que o tempo de implantação do bean torna-se menos importante. O alias espera um estado desinstalado até que o real bean aplique os triggers no mesmo.\n\t\n

Capítulo 7. Injeção de Dependência Avançada e IoC

Atualmente o *Dependency injection (DI)*, também chamado *Inversion of Control (IoC)*, fica no núcleo de muitos frameworks que adotam a noção de um container ou modelo de componente. Os modelos de componente estão descritos num capítulo anterior. O JBoss JMX Kernel, precursor do Microcontainer, fornecia apenas suporte DI/IoC de carga leve, primeiramente devido às limitações de acesso aos MBeans através do servidor MBeans. No entanto, com o novo modelo de componente baseado no POJO, diversos recursos novos e interessantes estão disponíveis.

Este capítulo apresenta como você pode aplicar os conceitos DI com a ajuda do JBoss Microcontainer. Estes conceitos serão expressados através do código XML, mas você pode também aplicar a maioria destes recursos usando anotações.

7.1. Fábrica de valor

A fábrica de valor é um bean que possui um ou mais métodos voltados à geração de valores para você. Consulte o Exemplo 7.1, "Fábrica de valor".

Exemplo 7.1. Fábrica de valor

```
<bean name="Binding" class="org.jboss.demos.ioc.vf.PortBindingManager">
  <constructor>
    <parameter>
      <map keyClass="java.lang.String" valueClass="java.lang.Integer">
 <entry>
   <key>http</key>
   <value>80</value>
 </entry>
 <entry>
   <key>ssh</key>
   <value>22</value>
 </entry>
      </map>
    </parameter>
  </constructor>
</bean>
<bean name="PortsConfig" class="org.jboss.demos.ioc.vf.PortsConfig">
  <property name="http"><value-factory bean="Binding" method="getPort"
parameter="http"/></property>
  <property name="ssh"><value-factory bean="Binding" method="getPort"
parameter="ssh"/></property>
  <property name="ftp">
    <value-factory bean="Binding" method="getPort">
      <parameter>ftp</parameter>
      <parameter>21</parameter>
    </value-factory>
  </property>
  <property name="mail">
    <value-factory bean="Binding" method="getPort">
      <parameter>mail</parameter>
      <parameter>25</parameter>
    </value-factory>
  </property>
</bean>
```

O Exemplo 7.2, "PortsConfig" apresenta como o PortsConfig bean usa o Binding bean para obter seus

valores através da invocação de método **getPort**.

Exemplo 7.2. PortsConfig

```
public class PortBindingManager {
    private Map<String, Integer> bindings;
    public PortBindingManager(Map<String, Integer> bindings)
    {
 this.bindings = bindings;
    }
    public Integer getPort(String key)
    {
 return getPort(key, null);
    }
    public Integer getPort(String key, Integer defaultValue)
    {
 if (bindings == null)
     return defaultValue;
 Integer value = bindings.get(key);
 if (value != null)
     return value;
 if (defaultValue != null)
     bindings.put(key, defaultValue);
 return defaultValue;
    }
}
```

7.2. Retornos de Chamada

O descritor apresentado no Exemplo 7.3, “Retornos de chamada para Coletar e Filtrar Beans” permite você coletar todos os beans de um certo tipo e ainda um certo limite de número de combinação de beans.

Em conjunção com o descritor no Exemplo 7.3, “Retornos de chamada para Coletar e Filtrar Beans”, o código Java apresentado no Exemplo 7.4, “Um Analisador para Coletar todos os Editores” demonstra um **Parser** que colecta todos os **Editor**s.

Exemplo 7.3. Retornos de chamada para Coletar e Filtrar Beans

```
<bean name="checker" class="org.jboss.demos.ioc.callback.Checker">
  <constructor>
    <parameter>
      <value-factory bean="parser" method="parse">
 <parameter>
   <array elementClass="java.lang.Object">
     <value>http://www.jboss.org</value>
     <value>SI</value>
     <value>3.14</value>
     <value>42</value>
   </array>
 </parameter>
      </value-factory>
    </parameter>
  </constructor>
</bean>
<bean name="editorA" class="org.jboss.demos.ioc.callback.DoubleEditor"/>
<bean name="editorB" class="org.jboss.demos.ioc.callback.LocaleEditor"/>
<bean name="parser" class="org.jboss.demos.ioc.callback.Parser">
  <incallback method="addEditor" cardinality="4..n"/>
  <uncallback method="removeEditor"/>
</bean>
<bean name="editorC" class="org.jboss.demos.ioc.callback.LongEditor"/>
<bean name="editorD" class="org.jboss.demos.ioc.callback.URLEditor"/>
```

Exemplo 7.4. Um Analisador para Coletar todos os Editores

```
public class Parser {
    private Set<Editor> editors = new HashSet<Editor>();
    ...
 public void addEditor(Editor editor)
    {
 editors.add(editor);
    }
    public void removeEditor(Editor editor)
    {
 editors.remove(editor);
    }
}
```

Perceba que o **incallback** e **uncallback** usam o mesmo nome de método para combinação.

```
<incallback method="addEditor" cardinality="4..n"/>
<uncallback method="removeEditor"/>
```

Um limite mínimo controla quantos editores podem ajudar o bean a melhorar do estado Configurado: **cardinality=4..n/>**

Normalmente, o **Checker** é criado e verifica o analisador. Isto está ilustrado no Exemplo 7.5, “O Verificador do Analisador”.

Exemplo 7.5. O Verificador do Analisador

```
public void create() throws Throwable {
    Set<String> strings = new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
    for (Object element : elements)
 strings.add(element.toString());
    if (expected.equals(strings) == false)
 throw new IllegalArgumentException("Illegal expected set: " + expected + "!=" +
strings);
}
```

7.3. Modo de Acesso do Bean

Os campos do bean não foram inspecionados com o BeanAccessMode padrão. No entanto, caso você especifique um BeanAccessMode diferente, os campos são acessados como parte das propriedades do bean. Consulte o Exemplo 7.6, "Definições Possíveis do BeanAccessMode", Exemplo 7.7, "Configuração do BeanAccessMode" e Exemplo 7.8, "Classe **FieldsBean**" para implementação.

Exemplo 7.6. Definições Possíveis do BeanAccessMode

```
public enum BeanAccessMode {
    STANDARD(BeanInfoCreator.STANDARD), // Getters and Setters
 FIELDS(BeanInfoCreator.FIELDS), // Getters/Setters and fields without getters
and setters
 ALL(BeanInfoCreator.ALL); // As above but with non public fields included
}
```

No caso abaixo um valor de String é configurado a um campo privado do String:

Exemplo 7.7. Configuração do BeanAccessMode

```
<bean name="FieldsBean" class="org.jboss.demos.ioc.access.FieldsBean" access-
mode="ALL">
  <property name="string">InternalString</property>
</bean>
```

Exemplo 7.8. Classe FieldsBean

```
public class FieldsBean {
    private String string;
    public void start()
    {
 if (string == null)
     throw new IllegalArgumentException("Strings should be set!");
    }
}
```

7.4. Bean Alias

Cada bean pode possuir qualquer número de aliases. Uma vez que os nomes do componente são tratados como Objetos, o tipo de alias não é limitado. Por padrão, a substituição de propriedade do sistema não é realizada. Você precisa configurar claramente o sinalizador substituído, conforme apresentado no Exemplo 7.9, “Bean Alias Simples”.

Exemplo 7.9. Bean Alias Simples

```
<bean name="SimpleName" class="java.lang.Object">
  <alias>SimpleAlias</alias>
  <alias replace="true">${some.system.property}</alias>
  <alias class="java.lang.Integer">12345</alias>
  <alias><javabean xmlns="urn:jboss:javabean:2.0"
class="org.jboss.demos.bootstrap.Main"/></alias>
</bean>
```

7.5. Suporte de Anotações XML (ou MetaData)

O suporte AOP é um recurso primário no JBoss Container. Você pode usar os aspectos AOP e beans planos em qualquer combinação. O Exemplo 7.10, “Intercepção de um Método baseado na Anotação” tenta interceptar uma invocação do método baseando-se em outra invocação. A anotação pode vir de qualquer lugar. Pode ser uma anotação de classe verdadeira ou uma anotação adicionada através da configuração xml.

Exemplo 7.10. Intercepção de um Método baseado na Anotação

```
<interceptor xmlns="urn:jboss:aop-beans:1.0" name="StopWatchInterceptor"
class="org.jboss.demos.ioc.annotations.StopWatchInterceptor"/>

<bind xmlns="urn:jboss:aop-beans:1.0" pointcut="execution(*
@org.jboss.demos.ioc.annotations.StopWatchLog->*(..)) OR execution(* *-
>@org.jboss.demos.ioc.annotations.StopWatchLog(..))">
  <interceptor-ref name="StopWatchInterceptor"/>
</bind>
</interceptor>
```

```
public class StopWatchInterceptor implements Interceptor {
    ...
 public Object invoke(Invocation invocation) throws Throwable
    {
 Object target = invocation.getTargetObject();
 long time = System.currentTimeMillis();
 log.info("Invocation [" + target + "] start: " + time);
 try
     {
  return invocation.invokeNext();
     }
 finally
     {
  log.info("Invocation [" + target + "] time: " + (System.currentTimeMillis() -
time));
     }
    }
}
```

O Exemplo 7.11, “Executor anotado pela classe verdadeira” e o Exemplo 7.12, “Executor simples com anotação XML” apresentam algumas maneiras diferentes de implementar executores.

Exemplo 7.11. Executor anotado pela classe verdadeira

```
<bean name="AnnotatedExecutor"
class="org.jboss.demos.ioc.annotations.AnnotatedExecutor">
```

```
public class AnnotatedExecutor implements Executor {
    ...
 @StopWatchLog // <-- Pointcut match!
     public void execute() throws Exception {
 delegate.execute();
    }
}
```

Exemplo 7.12. Executor simples com anotação XML

```
<bean name="SimpleExecutor"
class="org.jboss.demos.ioc.annotations.SimpleExecutor">
  <annotation>@org.jboss.demos.ioc.annotations.StopWatchLog</annotation> // <--
Pointcut match!
</bean>
```

```
public class SimpleExecutor implements Executor {
    private static Random random = new Random();
    public void execute() throws Exception
    {
 Thread.sleep(Math.abs(random.nextLong() % 101));
    }
}
```

Após adicionar os beans do chamador do executor, você poderá observar os executores em ação durante o trabalho, buscando pelo resultado do log tal como: Exemplo 7.13, “Resultado do Log Executor”.

Exemplo 7.13. Resultado do Log Executor

```
JBoss-MC-Demo INFO [15-12-2008 13:57:39] StopWatch - Invocation
[org.jboss.demos.ioc.annotations.AnnotatedExecutor@4d28c7] start: 1229345859234
JBoss-MC-Demo INFO [15-12-2008 13:57:39] StopWatch - Invocation
[org.jboss.demos.ioc.annotations.AnnotatedExecutor@4d28c7] time: 31
JBoss-MC-Demo INFO [15-12-2008 13:57:39] StopWatch - Invocation
[org.jboss.demos.ioc.annotations.SimpleExecutor@1b044df] start: 1229345859265
JBoss-MC-Demo INFO [15-12-2008 13:57:39] StopWatch - Invocation
[org.jboss.demos.ioc.annotations.SimpleExecutor@1b044df] time: 47

\t
```

7.6. Autowire

Autowire, ou injeção contextual, é um recurso comum dos frameworks IoC. O Exemplo 7.14, “Inclusão e Exclusão com o Autowire” apresenta como usar ou executar beans com o autowire.

Exemplo 7.14. Inclusão e Exclusão com o Autowire

```
<bean name="Square" class="org.jboss.demos.ioc.autowire.Square" autowire-
candidate="false"/>
<bean name="Circle" class="org.jboss.demos.ioc.autowire.Circle"/>
<bean name="ShapeUser" class="org.jboss.demos.ioc.autowire.ShapeUser">
  <constructor>
    <parameter><inject/></parameter>
  </constructor>
</bean>
<bean name="ShapeHolder" class="org.jboss.demos.ioc.autowire.ShapeHolder">
  <incallback method="addShape"/>
  <uncallback method="removeShape"/>
</bean>
<bean name="ShapeChecker" class="org.jboss.demos.ioc.autowire.ShapesChecker"/>
```

Em ambos os casos - *ShapeUser* e *ShapeChecker* - apenas *Circle* deverá ser usado, uma vez que o *Square* é excluído no binding contextual.

7.7. Fábrica de Bean

Quando você desejar mais de uma instância de um bean em particular, você precisará usar o padrão de fábrica do bean. O trabalho do Microcontainer é configurar e instalar a fábrica de bean como se ele fosse um bean simples. Depois disto, você precisa chamar o método **`createBean`** da fábrica de bean.

Por padrão, o Microcontainer cria uma instância **`GenericBeanFactory`**, mas você pode configurar sua própria fábrica. A única limitação é que os ganchos de configuração a assinatura são parecidos àquele do **`AbstractBeanFactory`**.

Exemplo 7.15. Fábrica de Bean Genérica

```
<bean name="Object" class="java.lang.Object"/>
<beanfactory name="DefaultPrototype"
class="org.jboss.demos.ioc.factory.Prototype">
  <property name="value"><inject bean="Object"/></property>
</beanfactory>
<beanfactory name="EnhancedPrototype"
class="org.jboss.demos.ioc.factory.Prototype"
factoryClass="org.jboss.demos.ioc.factory.EnhancedBeanFactory">
  <property name="value"><inject bean="Object"/></property>
</beanfactory>
<beanfactory name="ProxiedPrototype"
class="org.jboss.demos.ioc.factory.UnmodifiablePrototype"
factoryClass="org.jboss.demos.ioc.factory.EnhancedBeanFactory">
  <property name="value"><inject bean="Object"/></property>
</beanfactory>
<bean name="PrototypeCreator"
class="org.jboss.demos.ioc.factory.PrototypeCreator">
  <property name="default"><inject bean="DefaultPrototype"/></property>
  <property name="enhanced"><inject bean="EnhancedPrototype"/></property>
  <property name="proxied"><inject bean="ProxiedPrototype"/></property>
</bean>
```

Consulte o Exemplo 7.16, “BeanFactory Extendido” para uso de um BeanFactory extendido.

Exemplo 7.16. BeanFactory Extendido

```
public class EnhancedBeanFactory extends GenericBeanFactory {
    public EnhancedBeanFactory(KernelConfigurator configurator)
    {
 super(configurator);
    }
    public Object createBean() throws Throwable
    {
 Object bean = super.createBean();
 Class clazz = bean.getClass();
 if (clazz.isAnnotationPresent(SetterProxy.class))
     {
  Set<Class> interfaces = new HashSet<Class>();
  addInterfaces(clazz, interfaces);
  return Proxy.newProxyInstance(
           clazz.getClassLoader(),
           interfaces.toArray(new Class[interfaces.size()]),
           new SetterInterceptor(bean)
           );
     }
 else
     {
  return bean;
     }
    }
    protected static void addInterfaces(Class clazz, Set<Class> interfaces)
    {
 if (clazz == null)
     return;
 interfaces.addAll(Arrays.asList(clazz.getInterfaces()));
 addInterfaces(clazz.getSuperclass(), interfaces);
    }
    private class SetterInterceptor implements InvocationHandler
    {
 private Object target;
 private SetterInterceptor(Object target)
 {
     this.target = target;
 }
 public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable
 {
     String methodName = method.getName();
     if (methodName.startsWith("set"))
  throw new IllegalArgumentException("Cannot invoke setters.");
     return method.invoke(target, args);
 }
    }
}
```

```
public class PrototypeCreator {
    ...
 public void create() throws Throwable
    {
 ValueInvoker vi1 = (ValueInvoker)bfDefault.createBean();
 vi1.setValue("default");
 ValueInvoker vi2 = (ValueInvoker)enhanced.createBean();
 vi2.setValue("enhanced");
 ValueInvoker vi3 = (ValueInvoker)proxied.createBean();
 try
     {
  vi3.setValue("default");
  throw new Error("Should not be here.");
     }
 catch (Exception ignored)
     {
     }
    }
```

7.8. Construtor de Metadados de Bean

Quando usando o Microcontainer em seu código, use o **`BeanMetaDataBuilder`** para criar e configurar os metadados do bean.

Exemplo 7.17. BeanMetaDataBuilder

```
<bean name="BuilderUtil" class="org.jboss.demos.ioc.builder.BuilderUtil"/>
<bean name="BuilderExampleHolder"
class="org.jboss.demos.ioc.builder.BuilderExampleHolder">
  <constructor>
    <parameter><inject bean="BUExample"/></parameter>
  </constructor>
</bean>
```

A partir deste processo, você não expõe seu código aos detalhes de implementação do Microcontainer.

```
public class BuilderUtil {
    private KernelController controller;
    @Constructor
 public BuilderUtil(@Inject(bean = KernelConstants.KERNEL_CONTROLLER_NAME)
KernelController controller) {
 this.controller = controller;
    }
    public void create() throws Throwable {
 BeanMetaDataBuilder builder = BeanMetaDataBuilder.createBuilder("BUExample",
BuilderExample.class.getName());
 builder.addStartParameter(Kernel.class.getName(),
builder.createInject(KernelConstants.KERNEL_NAME));
 controller.install(builder.getBeanMetaData());
    }
    public void destroy() {
 controller.uninstall("BUExample");
    }
}
```

7.9. ClassLoader Personalizado

Você pode definir um ClassLoader personalizado por bean no Microcontainer. Quando definindo um classloader para toda a implantação, certifique-se de não criar uma dependência cíclica, por exemplo: um classloader recentemente definido que depende no mesmo.

Exemplo 7.18. Definição do ClassLoader por Bean

```
<classloader><inject bean="custom-classloader:0.0.0"/></classloader>
<!-- this will be explained in future article -->
<classloader name="custom-classloader" xmlns="urn:jboss:classloader:1.0" export-
all="NON_EMPTY" import-all="true"/>
<bean name="CustomCL" class="org.jboss.demos.ioc.classloader.CustomClassLoader">
  <constructor>
    <parameter><inject bean="custom-classloader:0.0.0"/></parameter>
  </constructor>
  <property name="pattern">org\.jboss\.demos\.ioc\..+</property>
</bean>
<bean name="CB1" class="org.jboss.demos.ioc.classloader.CustomBean"/>
<bean name="CB2" class="org.jboss.demos.ioc.classloader.CustomBean">
  <classloader><inject bean="CustomCL"/></classloader>
</bean>
```

O Exemplo 7.19, “Teste do ClassLoader Personalizado” apresenta um teste para verificar de que o CB2 usa um ClassLoader, que limita o escopo do pacote recarregável.

Exemplo 7.19. Teste do ClassLoader Personalizado

```
public class CustomClassLoader extends ClassLoader {
    private Pattern pattern;
    public CustomClassLoader(ClassLoader parent) {
 super(parent);
    }
    public Class<?> loadClass(String name) throws ClassNotFoundException {
 if (pattern == null || pattern.matcher(name).matches())
     return super.loadClass(name);
 else
     throw new ClassNotFoundException("Name '" + name + "' doesn't match pattern:
" + pattern);
    }
    public void setPattern(String regexp) {
 pattern = Pattern.compile(regexp);
    }
}
```

7.10. Modo Controlador

Por padrão, o Microcontainer usa o AUTO modo controlador. Ele empurra os beans tão distante quanto eles podem ir em relação às dependências. No entanto, existe outros dois modos: MANUAL e ON_DEMAND.

Caso o bean esteja marcado como ON_DEMAND, ele não seja usado ou instalado até que algum outro bean dependa explicitamente do mesmo. No modo MANUAL, o usuário Microcontainer deverá mover o bean adiante e de volta juntamente com o início do estado.

Exemplo 7.20. Modo Controlador do Bean

```
<bean name="OptionalService" class="org.jboss.demos.ioc.mode.OptionalService"
mode="On Demand"/>
<bean name="OptionalServiceUser"
class="org.jboss.demos.ioc.mode.OptionalServiceUser"/>
<bean name="ManualService" class="org.jboss.demos.ioc.mode.ManualService"
mode="Manual"/>
<bean name="ManualServiceUser"
class="org.jboss.demos.ioc.mode.ManualServiceUser">
  <start>
    <parameter><inject bean="ManualService" fromContext="context" state="Not
Installed"/></parameter>
  </start>
</bean>
```

Nota

Você pode injetar beans como também suas representações do componente não-modificado da representação do componente Microcontainer, usando o atributo fromContext da classe **`inject`**.

A revisão do código do **`OptionalServiceUser`** e **`ManualServiceUser`** serve para mostrar como usar o Microcontainer API para o manuseio do bean ON_DEMAND e MANUAL.

7.11. Ciclo

Os beans podem ser dependentes entre si ao invés de um ciclo. Por exemplo, *A* depende de *B* na construção, mas *B* depende de *A* no setter. Esta situação pode ser resolvida com facilidade uma vez que a separação do ciclo de vida do estado fino-granulado do Microcontainer.

Exemplo 7.21. Separação do Ciclo de Vida do Bean

```
<bean name="cycleA" class="org.jboss.demos.ioc.cycle.CyclePojo">
  <property name="dependency"><inject bean="cycleB"/></property>
</bean>
<bean name="cycleB" class="org.jboss.demos.ioc.cycle.CyclePojo">
  <constructor><parameter><inject bean="cycleA"
state="Instantiated"/></parameter></constructor>
</bean>
<bean name="cycleC" class="org.jboss.demos.ioc.cycle.CyclePojo">
  <property name="dependency"><inject bean="cycleD"/></property>
</bean>
<bean name="cycleD" class="org.jboss.demos.ioc.cycle.CyclePojo">
  <property name="dependency"><inject bean="cycleC"
state="Instantiated"/></property>
</bean>
```

7.12. Suprimento e Demanda

Ás vezes, tal como uma injeção, a dependência entre dois beans talvez não seja aparentemente rápida.

Tais dependências devem ser expressadas de forma clara, conforme apresentado no Exemplo 7.22, “Uso do Código Estatístico”.

Exemplo 7.22. Uso do Código Estatístico

```
<bean name="TMDemand" class="org.jboss.demos.ioc.demandsupply.TMDemander">
  <demand>TM</demand>
</bean>
<bean name="SimpleTMSupply"
class="org.jboss.demos.ioc.demandsupply.SimpleTMSupplyer">
  <supply>TM</supply>
</bean>
```

7.13. Instalação

Uma vez que um bean move através de estágios diferentes, você deve invocar alguns métodos em outros beans ou no mesmo bean. O Exemplo 7.23, “Métodos de Chamada em Estados Diferentes” apresenta como **`Entry`** invoca os métodos **`add`** e **`removeEntry`** do **`RepositoryManager`** para registrar-se e desregistrar-se.

Exemplo 7.23. Métodos de Chamada em Estados Diferentes

```
<bean name="RepositoryManager"
class="org.jboss.demos.ioc.install.RepositoryManager">
  <install method="addEntry">
    <parameter><inject fromContext="name"/></parameter>
    <parameter><this/></parameter>
  </install>
  <uninstall method="removeEntry">
    <parameter><inject fromContext="name"/></parameter>
  </uninstall>
</bean>
<bean name="Entry" class="org.jboss.demos.ioc.install.SimpleEntry">
  <install bean="RepositoryManager" method="addEntry" state="Instantiated">
    <parameter><inject fromContext="name"/></parameter>
    <parameter><this/></parameter>
  </install>
  <uninstall bean="RepositoryManager" method="removeEntry" state="Configured">
    <parameter><inject fromContext="name"/></parameter>
  </uninstall>
</bean>
```

7.14. Lazy Mock

Você pode ter uma dependência num bean que é raramente usado, mas demora mais para ser configurado. Você pode usar o lazy mock do bean, demonstrado no Exemplo 7.24, “Lazy Mock”, para resolver esta dependência. Quando você precisar de fato de um bean, chame e use o bean de destinação, esperando que ele tenha sido instalado por eles.

Exemplo 7.24. Lazy Mock

```
<bean name="lazyA" class="org.jboss.demos.ioc.lazy.LazyImpl">
  <constructor>
    <parameter>
      <lazy bean="lazyB">
 <interface>org.jboss.demos.ioc.lazy.ILazyPojo</interface>
      </lazy>
    </parameter>
  </constructor>
</bean>
<bean name="lazyB" class="org.jboss.demos.ioc.lazy.LazyImpl">
  <constructor>
    <parameter>
      <lazy bean="lazyA">
 <interface>org.jboss.demos.ioc.lazy.ILazyPojo</interface>
      </lazy>
    </parameter>
  </constructor>
</bean>
<lazy name="anotherLazy" bean="Pojo" exposeClass="true"/>
<bean name="Pojo" class="org.jboss.demos.ioc.lazy.Pojo"/>
```

7.15. Ciclo de vida

Por padrão, o Microcontainer usa os métodos **`create`**, **`start`**, e **`destroy`** quando ele passa por diversos estágios. No entanto, você não precisa que o Microcontainer os chame. Devido a isto, um sinalizador ignore está disponível.

Exemplo 7.25. Ciclos de Vida do Bean

```
<bean name="FullLifecycleBean-3"
class="org.jboss.demos.ioc.lifecycle.FullLifecycleBean"/>
<bean name="FullLifecycleBean-2"
class="org.jboss.demos.ioc.lifecycle.FullLifecycleBean">
  <create ignored="true"/>
</bean>
<bean name="FullLifecycleBean-1"
class="org.jboss.demos.ioc.lifecycle.FullLifecycleBean">
  <start ignored="true"/>
</bean>
```

Capítulo 8. Sistema de Arquivo Virtual

A duplicação do código de recurso de manuseio é um problema comum para desenvolvedores. Na maioria dos casos, o código determina informação a respeito de um recurso particular que pode ser um arquivo, um diretório, ou, no caso de um JAR, um URL remoto. Outro problema de duplicação é o código para o processamento dos arquivos aninhados. O Exemplo 8.1, “Problema de Duplicação de Recurso” ilustra este problema.

Exemplo 8.1. Problema de Duplicação de Recurso

```
public static URL[] search(ClassLoader cl, String prefix, String suffix) throws
IOException {
    Enumeration[] e = new Enumeration[]{
 cl.getResources(prefix),
 cl.getResources(prefix + "MANIFEST.MF")
    };
    Set all = new LinkedHashSet();
    URL url;
    URLConnection conn;
    JarFile jarFile;
    for (int i = 0, s = e.length; i < s; ++i)
 {
     while (e[i].hasMoreElements())
  {
      url = (URL)e[i].nextElement();
      conn = url.openConnection();
      conn.setUseCaches(false);
      conn.setDefaultUseCaches(false);
      if (conn instanceof JarURLConnection)
   {
       jarFile = ((JarURLConnection)conn).getJarFile();
   }
      else
   {
       jarFile = getAlternativeJarFile(url);
   }
      if (jarFile != null)
   {
       searchJar(cl, all, jarFile, prefix, suffix);
   }
      else
   {
       boolean searchDone = searchDir(all, new
File(URLDecoder.decode(url.getFile(), "UTF-8")), suffix);
       if (searchDone == false)
    {
        searchFromURL(all, prefix, suffix, url);
    }
   }
  }
 }
    return (URL[])all.toArray(new URL[all.size()]);
}

private static boolean searchDir(Set result, File file, String suffix) throws
IOException
{
    if (file.exists() && file.isDirectory())
 {
     File[] fc = file.listFiles();
     String path;
     for (int i = 0; i < fc.length; i++)
  {
      path = fc[i].getAbsolutePath();
      if (fc[i].isDirectory())
   {
       searchDir(result, fc[i], suffix);
   }
      else if (path.endsWith(suffix))
   {
```

```
          result.add(fc[i].toURL());
      }
    }
        return true;
  }
      return false;
}
```

Existem também diversos problemas com bloqueamento de arquivo nos sistemas do Windows, forçando os desenvolvedores a copiar todos os arquivos hot-deployable a outra localização, prevenindo o bloqueamento dos mesmos nas pastas implantadas (que preveniria tanto a exclusão e o sistema de arquivo baseado na desimplantação). O bloqueamento do arquivo é um problema grande cuja única solução costumava ser centralizar todos os códigos de carregamentos de recursos em um único local.

O projeto *VFS* foi a solução criada para todos estes problemas. VFS é a abreviatura para *Virtual File System* (Sistema de Arquivo Virtual).

8.1. API Público de VFS

O VFS é usado para dois propósitos principais, conforme apresentados no Usos do VFS.

Usos do VFS

- navegação simples de recurso
- *API* (Application Programmer Interface - Interface de Programador do Aplicativo) padrão do visitante

Conforme mencionado anteriormente, o manuseio e navegação de recursos são complexos no JDK simples. Você deve sempre checar o tipo de recurso e estas checagens podem ser pesadas. O VFS abstrai recursos num tipo de recurso único, VirtualFile.

Exemplo 8.2. Tipo de Recurso VirtualFile

```
public class VirtualFile implements Serializable {
    /**
     * Get certificates.
     *
     * @return the certificates associated with this virtual file
     */
    Certificate[] getCertificates()

    /**
     * Get the simple VF name (X.java)
     *
     * @return the simple file name
     * @throws IllegalStateException if the file is closed
     */
 String getName()

    /**
     * Get the VFS relative path name (org/jboss/X.java)
     *
     * @return the VFS relative path name
     * @throws IllegalStateException if the file is closed
     */
 String getPathName()

    /**
     * Get the VF URL (file://root/org/jboss/X.java)
     *
     * @return the full URL to the VF in the VFS.
     * @throws MalformedURLException if a url cannot be parsed
     * @throws URISyntaxException if a uri cannot be parsed
     * @throws IllegalStateException if the file is closed
     */
 URL toURL() throws MalformedURLException, URISyntaxException

    /**
     * Get the VF URI (file://root/org/jboss/X.java)
     *
     * @return the full URI to the VF in the VFS.
     * @throws URISyntaxException if a uri cannot be parsed
     * @throws IllegalStateException if the file is closed
     * @throws MalformedURLException for a bad url
     */
      URI toURI() throws MalformedURLException, URISyntaxException

    /**
     * When the file was last modified
     *
     * @return the last modified time
     * @throws IOException for any problem accessing the virtual file system
     * @throws IllegalStateException if the file is closed
     */
      long getLastModified() throws IOException

    /**
     * Returns true if the file has been modified since this method was last
called
     * Last modified time is initialized at handler instantiation.
     *
     * @return true if modifed, false otherwise
     * @throws IOException for any error
```

```
     */
    boolean hasBeenModified() throws IOException

   /**
    * Get the size
    *
    * @return the size
    * @throws IOException for any problem accessing the virtual file system
    * @throws IllegalStateException if the file is closed
    */
    long getSize() throws IOException

   /**
    * Tests whether the underlying implementation file still exists.
    * @return true if the file exists, false otherwise.
    * @throws IOException - thrown on failure to detect existence.
    */
    boolean exists() throws IOException

   /**
    * Whether it is a simple leaf of the VFS,
    * i.e. whether it can contain other files
    *
    * @return true if a simple file.
    * @throws IOException for any problem accessing the virtual file system
    * @throws IllegalStateException if the file is closed
    */
    boolean isLeaf() throws IOException

   /**
    * Is the file archive.
    *
    * @return true if archive, false otherwise
    * @throws IOException for any error
    */
    boolean isArchive() throws IOException

   /**
    * Whether it is hidden
    *
    * @return true when hidden
    * @throws IOException for any problem accessing the virtual file system
    * @throws IllegalStateException if the file is closed
    */
    boolean isHidden() throws IOException

   /**
    * Access the file contents.
    *
    * @return an InputStream for the file contents.
    * @throws IOException for any error accessing the file system
    * @throws IllegalStateException if the file is closed
    */
    InputStream openStream() throws IOException

   /**
    * Do file cleanup.
    *
    * e.g. delete temp files
    */
```

```
   void cleanup()

  /**
   * Close the file resources (stream, etc.)
   */
   void close()

  /**
   * Delete this virtual file
   *
   * @return true if file was deleted
   * @throws IOException if an error occurs
   */
   boolean delete() throws IOException

  /**
   * Delete this virtual file
   *
   * @param gracePeriod max time to wait for any locks (in milliseconds)
   * @return true if file was deleted
   * @throws IOException if an error occurs
   */
   boolean delete(int gracePeriod) throws IOException

  /**
   * Get the VFS instance for this virtual file
   *
   * @return the VFS
   * @throws IllegalStateException if the file is closed
   */
   VFS getVFS()

  /**
   * Get the parent
   *
   * @return the parent or null if there is no parent
   * @throws IOException for any problem accessing the virtual file system
   * @throws IllegalStateException if the file is closed
   */
   VirtualFile getParent() throws IOException

  /**
   * Get a child
   *
   * @param path the path
   * @return the child or <code>null</code> if not found
   * @throws IOException for any problem accessing the VFS
   * @throws IllegalArgumentException if the path is null
   * @throws IllegalStateException if the file is closed or it is a leaf node
   */
   VirtualFile getChild(String path) throws IOException

  /**
   * Get the children
   *
   * @return the children
   * @throws IOException for any problem accessing the virtual file system
   * @throws IllegalStateException if the file is closed
   */
   List<VirtualFile> getChildren() throws IOException
```

```
    /**
     * Get the children
     *
     * @param filter to filter the children
     * @return the children
     * @throws IOException for any problem accessing the virtual file system
     * @throws IllegalStateException if the file is closed or it is a leaf node
     */
      List<VirtualFile> getChildren(VirtualFileFilter filter) throws IOException

    /**
     * Get all the children recursively<p>
     *
     * This always uses {@link VisitorAttributes#RECURSE}
     *
     * @return the children
     * @throws IOException for any problem accessing the virtual file system
     * @throws IllegalStateException if the file is closed
     */
      List<VirtualFile> getChildrenRecursively() throws IOException

    /**
     * Get all the children recursively<p>
     *
     * This always uses {@link VisitorAttributes#RECURSE}
     *
     * @param filter to filter the children
     * @return the children
     * @throws IOException for any problem accessing the virtual file system
     * @throws IllegalStateException if the file is closed or it is a leaf node
     */
      List<VirtualFile> getChildrenRecursively(VirtualFileFilter filter) throws
IOException

    /**
     * Visit the virtual file system
     *
     * @param visitor the visitor
     * @throws IOException for any problem accessing the virtual file system
     * @throws IllegalArgumentException if the visitor is null
     * @throws IllegalStateException if the file is closed
     */
      void visit(VirtualFileVisitor visitor) throws IOException
      }
```

Todas as operações de Sistema de Arquivo apenas de leitura estão disponíveis, além de poucas opções de limpeza ou exclusão do recurso. O manuseio de exclusão e limpeza é necessário quando manuseando com alguns arquivos temporários, tais como os arquivos criados para o manuseio aninhado em jars.

Para alterar do manuseio de recurso File ou URL do JDK para o novo VirtualFile, você precisará de um VirtualFile raiz, que é fornecido pela classe **VFS**, com a ajuda do URL ou parâmetro URI.

Exemplo 8.3. Uso da Classe VFS

```
public class VFS {
    /**
     * Get the virtual file system for a root uri
     *
     * @param rootURI the root URI
     * @return the virtual file system
     * @throws IOException if there is a problem accessing the VFS
     * @throws IllegalArgumentException if the rootURL is null
     */
    static VFS getVFS(URI rootURI) throws IOException

    /**
     * Create new root
     *
     * @param rootURI the root url
     * @return the virtual file
     * @throws IOException if there is a problem accessing the VFS
     * @throws IllegalArgumentException if the rootURL
     */
    static VirtualFile createNewRoot(URI rootURI) throws IOException

    /**
     * Get the root virtual file
     *
     * @param rootURI the root uri
     * @return the virtual file
     * @throws IOException if there is a problem accessing the VFS
     * @throws IllegalArgumentException if the rootURL is null
     */
    static VirtualFile getRoot(URI rootURI) throws IOException

    /**
     * Get the virtual file system for a root url
     *
     * @param rootURL the root url
     * @return the virtual file system
     * @throws IOException if there is a problem accessing the VFS
     * @throws IllegalArgumentException if the rootURL is null
     */
    static VFS getVFS(URL rootURL) throws IOException

    /**
     * Create new root
     *
     * @param rootURL the root url
     * @return the virtual file
     * @throws IOException if there is a problem accessing the VFS
     * @throws IllegalArgumentException if the rootURL
     */
    static VirtualFile createNewRoot(URL rootURL) throws IOException

    /**
     * Get the root virtual file
     *
     * @param rootURL the root url
     * @return the virtual file
     * @throws IOException if there is a problem accessing the VFS
     * @throws IllegalArgumentException if the rootURL
     */
    static VirtualFile getRoot(URL rootURL) throws IOException
```

```
    /**
     * Get the root file of this VFS
     *
     * @return the root
     * @throws IOException for any problem accessing the VFS
     */
    VirtualFile getRoot() throws IOException
}
```

Os três diferentes métodos são similares.

- **`getVFS`**
- **`createNewRoot`**
- **`getRoot`**

O **`getVFS`** retorna uma instância VFS mas não cria uma instância VirtualFile. Isto é importante uma vez que existem métodos que ajudam a configuração de uma instância VFS (consulte javadocs do API de classes VFS) antes de instruí-la a criar uma raiz do VirtualFile.

Por outro lado, os dois outros métodos usam configurações para a criação da raiz. A diferença entre **`createNewRoot`** e **`getRoot`** são os detalhes de cache, dos quais serão descritos mais tarde.

Exemplo 8.4. Uso do `getVFS`

```
URL rootURL = ...; // get root url
VFS vfs = VFS.getVFS(rootURL);
// configure vfs instance
VirtualFile root1 = vfs.getRoot();
// or you can get root directly
VirtualFile root2 = VFS.crateNewRoot(rootURL);
VirtualFile root3 = VFS.getRoot(rootURL);
```

Outro detalhe útil do VFS API é sua implantação de um padrão de visitante apropriado. É muito simples coletar recursivamente recursos diferentes, uma tarefa que é difícil realizar com o carregamento do recurso do JDK simples.

Exemplo 8.5. Recursos de Coleção Recursiva

```
public interface VirtualFileVisitor {
    /**
     * Get the search attribues for this visitor
     *
     * @return the attributes
     */
    VisitorAttributes getAttributes();

    /**
     * Visit a virtual file
     *
     * @param virtualFile the virtual file being visited
     */
    void visit(VirtualFile virtualFile);
}

VirtualFile root = ...; // get root
VirtualFileVisitor visitor = new SuffixVisitor(".class"); // get all classes
root.visit(visitor);
```

8.2. Arquitetura VFS

Embora o API público seja bastante intuitivo, os detalhes de implantação real adicionam complexidade. Alguns conceitos precisam ser explicados em maiores detalhes.

Toda vez que você cria uma instância VFS, sua instância de combinação **`VFSContext`** é criada. A criação é feita através do **`VFSContextFactory`**. Protocolos diferentes mapeiam instâncias **`VFSContextFactory`** diferentes. por exemplo, o **`file/vfsfile`** mapeia para o **`FileSystemContextFactory`**, enquanto o **`zip/vfszip`** mapeia para o **`ZipEntryContextFactory`**.

Toda vez que uma instância **`VirtualFile`** é criada, sua combinação VirtualFileHandler é criada. Esta instância VirtualFileHandler sabe como manusear diferentes tipos de recursos apropriados. O **`VirtualFile`** API apenas delega invocações para a própria referência VirtualFileHandler.

A instância **`VFSContext`** sabe como criar as intâncias VirtualFileHandler de acordo com o tipo de recurso. Por exemplo, o ZipEntryContextFactory cria o **`ZipEntryContext`**, que cria o **`ZipEntryHandler`**.

8.3. Implantações Existentes

A partir dos arquivos, diretórios (FileHandler) e arquivos (ZipEntryHandler) o Microcontainer também suporta outros casos de uso avançados. O primeiro é o *Assembled*, que é parecido ao que o Eclipse chama de *Linked Resources*. Seu propósito é obter recursos existentes de árvores diferentes e "simulá-los" em árvores de recurso único.

Exemplo 8.6. Implementação dos VirtualFileHandlers Montados

```
AssembledDirectory sar =
AssembledContextFactory.getInstance().create("assembled.sar");

URL url = getResource("/vfs/test/jar1.jar");
VirtualFile jar1 = VFS.getRoot(url);
sar.addChild(jar1);

url = getResource("/tmp/app/ext.jar");
VirtualFile ext1 = VFS.getRoot(url);
sar.addChild(ext);

AssembledDirectory metainf = sar.mkdir("META-INF");

url = getResource("/config/jboss-service.xml");
VirtualFile serviceVF = VFS.getRoot(url);
metainf.addChild(serviceVF);

AssembledDirectory app = sar.mkdir("app.jar");
url = getResource("/app/someapp/classes");
VirtualFile appVF = VFS.getRoot(url);
app.addPath(appVF, new SuffixFilter(".class"));
```

Outra implementação é dos arquivos *in-memory*. Esta implementação nasceu da necessidade de manusear facilmente os bytes gerados pelo AOP. Ao invés de usar os arquivos temporários, você pode soltar bytes no VirtualFileHandlers em memória.

Exemplo 8.7. Implementação dos VirtualFileHandlers em memória

```
URL url = new URL("vfsmemory://aopdomain/org/acme/test/Test.class");
byte[] bytes = ...; // some AOP generated class bytes
MemoryFileFactory.putFile(url,  bytes);

VirtualFile classFile = VFS.getVirtualFile(new URL("vfsmemory://aopdomain"),
"org/acme/test/Test.class");
InputStream bis = classFile.openStream(); // e.g. load class from input stream
```

8.4. Ganchos de Extensão

É fácil de estender o VFS com um novo protocolo, simular o que nós fizemos com o **`Assembled`** e **`Memory`**. Tudo o que você precisa fazer é uma combinação das implementações **`VFSContexFactory`**, **`VFSContext`**, **`VirtualFileHandler`**, **`FileHandlerPlugin`** e **`URLStreamHandler`**. O **`VFSContextFactory`** é trivial enquanto que outros dependem da complexidade de sua tarefa. Você pode implementar o acesso **`rar`**, **`tar`**, **`gzip`**, ou mesmo o acesso remoto.

Após implementação de um novo protocolo, registre o novo VFSContextFactory com o VFSContextFactoryLocator.

8.5. Recursos

Um dos primeiros problemas que os desenvolvedores do Microcontainer encontraram foi o uso próprio

de recursos aninhados, particularmente os arquivos jar aninhados. Por exemplo: as implantações ear normais: **gema.ear/ui.war/WEB-INF/lib/struts.jar**.

Nós temos duas opções para ler conteúdos do **struts.jar**:

- manuseio de recursos em memória
- criar cópias temporárias do nível superior de jars aninhados, de forma recursiva

A primeira opção é fácil de implantar, mas consume muita memória, necessitando de aplicativos potencialmente grandes residam na memória. A próxima abordagem deixa para trás um grande número de arquivos temporários, que devem ser invisíveis ao usuário final e, portanto, devem desaparecer após a desimplantação.

Considere o seguinte cenário: um usuário acessa a instância VFS URL, que aponta a um recurso aninhado.

A maneira com que o VFS deve manusear isto é recriar o caminho por completo a partir do rascunho: isto desempacotaria mais e mais os recursos aninhados. Isto leva a um número grande de arquivos temporários.

O Microcontainer evita isto pelo uso do **VFSRegistry**, **VFSCache** e **TempInfo**.

Quando você perguntar pelo VirtualFile sobre VFS (**getRoot** e não o **createNewRoot**), o VFS pergunta pela implementação **VFSRegistry** para fornecer o arquivo. O **DefaultVFSRegistry** existente checa primeiramente se existe uma raiz **VFSContext** existente para o URI fornecido. Caso existir, o **DefaultVFSRegistry** tenta primeiro navegar ao **TempInfo** existente (link para arquivos existentes), retornando à navegação regular caso tal arquivo temporário não existir. Desta maneira, você usará completamente quaisquer arquivos temporários que já foram desempacotados, economizando tempo e memória do disco. Caso nenhum **VFSContext** coincidente for encontrado no cache, o código criará uma nova entrada e continuará com a navegação padrão.

A determinação de como o **VFSCache** manuseia as entradas do VFSContext com cache depende na implantação usada. O **VFSCache** é configurado através do **VFSCacheFactory**. Por padrão, nada estará com cache, mas existem algumas implementações **VFSCache** usando algoritmos tais como **Least Recently Used (LRU)** ou **timed cache**.

Capítulo 9. A Camada ClassLoading

O JBoss possui uma maneira única de lidar com o classloading, assim como a camada de classloading que vem com o Microcontainer. O Classloading é um suplemento que você pode usar quando desejar classloading sem padrão. As alterações à camada do ClassLoading do EAP 5.1 são úteis e em tempo certo com o aumento da demanda do classloading de estilo OSGi e um número de novas especificações do classloading do Java à vista.

A camada do Microcontainer ClassLoading é uma camada de abstração. A maioria de detalhes estão bloqueados em métodos privados de pacote e privados sem comprometer a extensão e funcionalidade disponível através das classes públicas que fazem o API. Isto significa que você pode codificar em referência à política, mas não em referência aos detalhes do classloader.

O projeto ClassLoader está dividido em 3 sub-projetos

- classloader
- classloading
- classloading-vfs

O **`classloader`** contém uma extensão **`java.lang.ClassLoader`** personalizada sem qualquer política classloading específica. A política classloading inclui o conhecimento de onde e como carregar.

O **`Classloading`** é uma extensão dos mecanismos de dependência do Microcontainer. Sua implementação VFS-backed é **`classloading-vfs`**. Consulte o Capítulo 8, *Sistema de Arquivo Virtual* para maiores informações sobre o VFS.

9.1. ClassLoader

A implantação **`ClassLoader`** suporta as políticas pugláveis, é a classe final, sem mencionar que não deve ser alterada. Para gravar suas implantações ClassLoader, grave um **`ClassLoaderPolicy`** que fornece um API mais simples para carregamento de classes e recursos, além de especificar outras regras associadas com o classloader.

Para personalizar o classloading, instancie o **`ClassLoaderPolicy`** e o registre com um **`ClassLoaderSystem`** para criar um **`ClassLoader`** personalizado. Você pode criar um **`ClassLoaderDomain`** para a partição do **`ClassLoaderSystem`**.

A camada **`ClassLoader`** também inclui a implementação de, por exemplo, modelo DelegateLoader, classloading, filtros de recurso e políticas de delegação pais-filhos.

O período de execução é o JMX ativado para expor a política usada para cada classloader. Ele também fornece as estatísticas do classloading e métodos de depuração para ajudar a determinar da onde vem o carregamento.

Exemplo 9.1. Classe `ClassLoaderPolicy`

O **`ClassLoaderPolicy`** controla a maneira com que o seu classloading trabalha.

```
public abstract class ClassLoaderPolicy extends BaseClassLoaderPolicy {
    public DelegateLoader getExported()

        public String[] getPackageNames()

        protected List<? extends DelegateLoader> getDelegates()

        protected boolean isImportAll()
        protected boolean isCacheable()
        protected boolean isBlackListable()

        public abstract URL getResource(String path);

    public InputStream getResourceAsStream(String path)

        public abstract void getResources(String name, Set<URL> urls) throws
IOException;

    protected ProtectionDomain getProtectionDomain(String className, String
path)
        public PackageInformation getPackageInformation(String packageName)
        public PackageInformation getClassPackageInformation(String className,
String packageName)

        protected ClassLoader isJDKRequest(String name)
        }
}
```

Segue abaixo dois exemplos do **`ClassLoaderPolicy`**. O primeiro restaura os recursos baseados em expressões regulares, enquanto que o segundo manuseia os recursos encriptados.

Exemplo 9.2. ClassLoaderPolicy com o Suporte de Expressão Regular

```
public class RegexpClassLoaderPolicy extends ClassLoaderPolicy {
    private VirtualFile[] roots;
    private String[] packageNames;

    public RegexpClassLoaderPolicy(VirtualFile[] roots)
    {
 this.roots = roots;
    }

    @Override
 public String[] getPackageNames()
    {
 if (packageNames == null)
     {
  Set<String> exportedPackages = PackageVisitor.determineAllPackages(roots,
null, ExportAll.NON_EMPTY, null, null, null);
  packageNames = exportedPackages.toArray(new String[exportedPackages.size()]);
     }
 return packageNames;
    }

    protected Pattern createPattern(String regexp)
    {
 boolean outside = true;
 StringBuilder builder = new StringBuilder();
 for (int i = 0; i < regexp.length(); i++)
     {
  char ch = regexp.charAt(i);
  if ((ch == '[' || ch == ']' || ch == '.') && escaped(regexp, i) == false)
      {
   switch (ch)
       {
       case '[' : outside = false; break;
       case ']' : outside = true; break;
       case '.' : if (outside) builder.append("\\"); break;
       }
      }

  builder.append(ch);
     }
 return Pattern.compile(builder.toString());
    }

    protected boolean escaped(String regexp, int i)
    {
 return i > 0 && regexp.charAt(i - 1) == '\\';
    }

    public URL getResource(String path)
    {
 Pattern pattern = createPattern(path);
 for (VirtualFile root : roots)
     {
  URL url = findURL(root, root, pattern);
  if (url != null)
      return url;
     }
 return null;
    }
```

```
   private URL findURL(VirtualFile root, VirtualFile file, Pattern pattern)
   {
try
    {
 String path = AbstractStructureDeployer.getRelativePath(root, file);
 Matcher matcher = pattern.matcher(path);
 if (matcher.matches())
     return file.toURL();

 List<VirtualFile> children = file.getChildren();
 for (VirtualFile child : children)
     {
  URL url = findURL(root, child, pattern);
  if (url != null)
      return url;
     }

 return null;
    }
catch (Exception e)
    {
 throw new RuntimeException(e);
    }
   }

   public void getResources(String name, Set<URL> urls) throws IOException
   {
Pattern pattern = createPattern(name);
for (VirtualFile root : roots)
    {
 RegexpVisitor visitor = new RegexpVisitor(root, pattern);
 root.visit(visitor);
 urls.addAll(visitor.getUrls());
    }
   }

   private static class RegexpVisitor implements VirtualFileVisitor
   {
private VirtualFile root;
private Pattern pattern;
private Set<URL> urls = new HashSet<URL>();

private RegexpVisitor(VirtualFile root, Pattern pattern)
{
    this.root = root;
    this.pattern = pattern;
}

public VisitorAttributes getAttributes()
{
    return VisitorAttributes.RECURSE_LEAVES_ONLY;
}

public void visit(VirtualFile file)
{
    try
 {
     String path = AbstractStructureDeployer.getRelativePath(root, file);
     Matcher matcher = pattern.matcher(path);
     if (matcher.matches())
```

```
    urls.add(file.toURL());
   }
      catch (Exception e)
   {
       throw new RuntimeException(e);
   }
  }

  public Set<URL> getUrls()
  {
      return urls;
  }
     }
}
```

O **RegexpClassLoaderPolicy** usa um mecanismo simples para encontrar recursos de combinação. As implantações do mundo real podem ser mais compreensivas e elegantes.

```
public class RegexpService extends PrintService {
    public void start() throws Exception
    {
  System.out.println();

  ClassLoader cl = getClass().getClassLoader();
  Enumeration<URL> urls = cl.getResources("config/[^.]+\\.[^.]{1,4}");
  while (urls.hasMoreElements())
      {
   URL url = urls.nextElement();
   print(url.openStream(), url.toExternalForm());
      }
    }
}
```

O serviço regexp usa a expressão regular padrão **config/[^.]+\\.[^.]{1,4}** para listar recursos sob o diretório **config/**/. O comprimento do sufixo é limitado de forma que os nomes do arquivo, tais como excluded.properties, serão ignorados.

Exemplo 9.3. ClassLoaderPolicy com Suporte de Criptografia

```
public class CrypterClassLoaderPolicy extends VFSClassLoaderPolicy {
    private Crypter crypter;

    public CrypterClassLoaderPolicy(String name, VirtualFile[] roots,
VirtualFile[] excludedRoots, Crypter crypter) {
        super(name, roots, excludedRoots);
        this.crypter = crypter;
    }

    @Override
    public URL getResource(String path) {
        try
            {
                URL resource = super.getResource(path);
                return wrap(resource);
            }
        catch (IOException e)
            {
                throw new RuntimeException(e);
            }
    }

    @Override
    public InputStream getResourceAsStream(String path) {
        InputStream stream = super.getResourceAsStream(path);
        return crypter.crypt(stream);
    }

    @Override
    public void getResources(String name, Set<URL> urls) throws IOException {
        super.getResources(name, urls);
        Set<URL> temp = new HashSet<URL>(urls.size());
        for (URL url : urls)
            {
                temp.add(wrap(url));
            }
        urls.clear();
        urls.addAll(temp);
    }

    protected URL wrap(URL url) throws IOException {
        return new URL(url.getProtocol(), url.getHost(), url.getPort(),
url.getFile(), new CrypterURLStreamHandler(crypter));
    }
}
```

O Exemplo 9.3, “ClassLoaderPolicy com Suporte de Criptografia” apresenta como criptografar os JARs. Você pode configurar quais recursos a serem criptografados, com exceção de conteúdos do diretório **META-INF/**.

```
public class EncryptedService extends PrintService {
    public void start() throws Exception
    {
 ClassLoader cl = getClass().getClassLoader();

 URL url = cl.getResource("config/settings.txt");
 if (url == null)
     throw new IllegalArgumentException("No such settings.txt.");

 InputStream is = url.openStream();
 print(is, "Printing settings:\n");

 is = cl.getResourceAsStream("config/properties.xml");
 if (is == null)
     throw new IllegalArgumentException("No such properties.xml.");

 print(is, "\nPrinting properties:\n");
    }
}
```

Este serviço emite os conteúdos dos dois arquivos de configuração. Ele demonstra que a descriptografia de quaisquer recursos criptografados está oculta na camada do classloading.

Para testá-lo, você pode tanto criptografar o módulo de política ou usar uma criptografia existente. Para colocar isto em prática, você precisa vincular apropriadamente o EncryptedService para **ClassLoaderSystem** e implantadores.

O particionamento do **ClassLoaderSystem** será discutido mais adiante neste capítulo.

9.2. ClassLoading

Ao invés de usar a abstração do **ClassLoader** diretamente, você pode criar os módulos do **ClassLoading** que contém declarações das dependências **ClassLoader**. Uma vez que as dependências são especificadas, os **ClassLoaderPolicy**s são construídos e conectados de acordo.

Para facilitar a definição do **ClassLoaders** antes deles de fato existirem, a abstração inclui um modelo **ClassLoadingMetaData**.

O **ClassLoadingMetaData** pode ser exposto assim como o Objetivo Gerenciado com o novo serviço de perfil do JBoss EAP. Isto ajuda os administradores de sistema a lidar com os detalhes de políticas mais abstratos ao invés dos detalhes de implantação.

Exemplo 9.4. ClassLoadingMetaData Exposto como um Objeto Gerenciado

```
public class ClassLoadingMetaData extends NameAndVersionSupport {
    /** The serialVersionUID */
    private static final long serialVersionUID = -2782951093046585620L;

    /** The classloading domain */
    private String domain;

    /** The parent domain */
    private String parentDomain;

    /** Whether to make a subdeployment classloader a top-level classloader */
    private boolean topLevelClassLoader = false;

    /** Whether to enforce j2se classloading compliance */
    private boolean j2seClassLoadingCompliance = true;

    /** Whether we are cacheable */
    private boolean cacheable = true;

    /** Whether we are blacklistable */
    private boolean blackListable = true;

    /** Whether to export all */
    private ExportAll exportAll;

    /** Whether to import all */
    private boolean importAll;

    /** The included packages */
    private String includedPackages;

    /** The excluded packages */
    private String excludedPackages;

    /** The excluded for export */
    private String excludedExportPackages;

    /** The included packages */
    private ClassFilter included;

    /** The excluded packages */
    private ClassFilter excluded;

    /** The excluded for export */
    private ClassFilter excludedExport;

    /** The requirements */
    private RequirementsMetaData requirements = new RequirementsMetaData();

    /** The capabilities */
    private CapabilitiesMetaData capabilities = new CapabilitiesMetaData();

    ... setters & getters
```

O Exemplo 9.5, “ClassLoading API definido no XML” e Exemplo 9.6, “ClassLoading API definido no Java” apresentam o ClassLoading API definido no XML e Java, respectivamente.

Exemplo 9.5. ClassLoading API definido no XML

```
<classloading xmlns="urn:jboss:classloading:1.0"
       name="ptd-jsf-1.0.war"
       domain="ptd-jsf-1.0.war"
       parent-domain="ptd-ear-1.0.ear"
       export-all="NON_EMPTY"
       import-all="true"
       parent-first="true"/>
```

Exemplo 9.6. ClassLoading API definido no Java

```
ClassLoadingMetaData clmd = new ClassLoadingMetaData();
if (name != null)
    clmd.setDomain(name + "_Domain");
clmd.setParentDomain(parentDomain);
clmd.setImportAll(true);
clmd.setExportAll(ExportAll.NON_EMPTY);
clmd.setVersion(Version.DEFAULT_VERSION);
```

Você pode adicionar o ClassLoadingMetaData à implantação tanto de forma programática ou declarativa, através do `jboss-classloading.xml`.

Exemplo 9.7. Adição do ClassLoadingMetaData usando o `jboss-classloading.xml`

```
<classloading xmlns="urn:jboss:classloading:1.0"
       domain="DefaultDomain"
       top-level-classloader="true"
       export-all="NON_EMPTY"
       import-all="true">
</classloading>
```

O DefaultDomain é compartilhado entre todos os aplicativos que não definem seus próprios domínios.

Exemplo 9.8. Isolação do Nível do Domínio Típico

```
<classloading xmlns="urn:jboss:classloading:1.0"
       domain="IsolatedDomain"
       export-all="NON_EMPTY"
       import-all="true">
</classloading>
```

Exemplo 9.9. Isolação com um Pai Específico

```
<classloading xmlns="urn:jboss:classloading:1.0"
       domain="IsolatedWithParentDomain"
       parent-domain="DefaultDomain"
       export-all="NON_EMPTY"
       import-all="true">
</classloading>
```

Exemplo 9.10. j2seClassLoadingCompliance sem conformidade

```
<classloading xmlns="urn:jboss:classloading:1.0"
       parent-first="false">
</classloading>
```

O **`.war`** implanta o uso deste método por padrão. Ao invés de realizar pesquisas do primeiro do pai padrão, você deve checar primeiramente seus próprios recursos.

Exemplo 9.11. Implantação OSGi Típica

```
<classloading xmlns="urn:jboss:classloading:1.0">
  <requirements>
    <package name="org.jboss.dependency.spi"/>
  </requirements>
  <capabilities>
    <package name="org.jboss.cache.api"/>
    <package name="org.jboss.kernel.spi"/>
  </capabilities>
</classloading>
```

Exemplo 9.12. Importação e Exportações de Bibliotecas e Módulos completos, ao invés de Pacotes de granulação fina.

```
<classloading xmlns="urn:jboss:classloading:1.0">
  <requirements>
    <module name="jboss-reflect.jar"/>
  </requirements>
  <capabilities>
    <module name="jboss-cache.jar"/>
  </capabilities>
</classloading>
```

```
<classloading xmlns="urn:jboss:classloading:1.0">
  <requirements>
    <package name="si.acme.foobar"/>
    <module name="jboss-reflect.jar"/>
  </requirements>
  <capabilities>
    <package name="org.alesj.cl"/>
    <module name="jboss-cache.jar"/>
  </capabilities>
</classloading>
```

Você pode misturar os tipos de solicitação e capacidades usando os pacotes e módulos.

O sub-projeto do classloading usa uma implantação padrão pequena recurso-visitante.

No projeto **`ClassLoader`** a conexão entre a implantação e o classloading é feita através da classe **`Module`** que mantém toda a informação para aplicar restrições apropriadamente no padrão do visitante, tal como filtramento.

Exemplo 9.13. Interfaces ResourceVisitor e ResourceContext

```
public interface ResourceVisitor {
    ResourceFilter getFilter();

    void visit(ResourceContext resource);
}

public interface ResourceContext {
    URL getUrl();

    ClassLoader getClassLoader();

    String getResourceName();

    String getClassName();

    boolean isClass();

    Class<?> loadClass();

    InputStream getInputStream() throws IOException;

    byte[] getBytes() throws IOException;
}
```

Para usar o módulo, instancie sua instância ResourceVisitor e passe-a para o método **`Module::visit`**. Este recurso é usado no framework para índice do uso das anotações nas implantações.

9.3. ClassLoading VFS

Estas amostras fornecem uma implantação **`ClassLoaderPolicy`** que usa um projeto do Sistema do Arquivo Virtual do JBoss para carregar classes e recursos. Você pode usar esta ideia diretamente ou em combinação com o framework classloading.

Opcionalmente, você pode definir seus módulos dentro da configuração do Microcontainer.

Exemplo 9.14. Classloading Module Deployer

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <classloader name="anys-classloader" xmlns="urn:jboss:classloader:1.0" import-
all="true" domain="Anys" parent-domain="DefaultDomain">
    <capabilities>
      <package name="org.jboss.test.deployers.vfs.reflect.support.web"/>
    </capabilities>
    <root>${jboss.tests.url}</root>
  </classloader>

  <bean name="AnyServlet"
class="org.jboss.test.deployers.vfs.reflect.support.web.AnyServlet">
    <classloader><inject bean="anys-classloader:0.0.0"/></classloader>
  </bean>

</deployment>
```

A classe **VFSClassLoaderFactory** transforma o implantador XML em um **VFSClassLoaderPolicyModule**, que então cria a instância **ClassLoader** atual. Você pode então usar esta nova instância **ClassLoader** com seus beans.

Nota

O **VFSClassLoaderFactory** estende o **ClassLoadingMetaData**, de forma que todas as amostras relativas ao **ClassLoadingMetaData** são também válidas nestas circunstâncias.

Capítulo 10. Framework de Implantação Virtual

O novo *Virtual Deployment Framework (VDF)* é uma maneira aprimorada para gerenciar as implantações no Microcontainer. Este capítulo detalha alguns de seus recursos úteis.

10.1. Manuseio Agnóstico de Tipos de Implantação

O tipo tradicional de uma implantação virtual é baseado em classes que já existem em espaço de classe compartilhada ou domínio. Neste caso, o final de cada produto é um novo serviço instalado no servidor a partir de um novo cliente principal. A maneira tradicional de realizar isto é realizar o upload de um arquivo descritor. O novo VDF simplifica este processo pela passagem sobre bytes e serializando-os em uma nova instância **`Deployment`**

O outro tipo de implantação, que estende o primeiro, é uma implantação baseada no sistema de arquivo básico, backed up pelo Microcontainer VFS. Esta abordagem está descrita em mais detalhes no Capítulo 8, *Sistema de Arquivo Virtual*.

10.2. A separação do Reconhecimento da Estrutura da lógica do ciclo de via da Implantação

Com o objetivo de realizar um trabalho real de implantação, você deve primeiramente entender suas estruturas, incluindo suas localizações de metadados e classpaths.

As localizações de metadados incluem os arquivos de configuração tais como **`my-jboss-beans.xml`**, **`web.xml`**, **`ejb-jar.xml`**. Os classloader são raízes do classloader, tais como **`WEB-INF/classes`** ou **`myapp.ear/lib`**.

Você pode proceder com o manuseamento de implantação atual, tendo a estrutura em mente.

Típico Ciclo de vida de Implantação

1. O **`MainDeployer`** passa a implantação para determinar o conjunto de **`StructuralDeployer`**s para reconhecimento e receber novamente o contexto de Implantação.
2. Em seguida, o **`MainDeployer`** passa o contexto de Implantação resultante ao **`Deployers`** para manuseio do **`Deployer`** apropriado.

Desta maneira, o MainDeployer é um agente com a responsabilidade de decidir quais Implantadores usar.

No caso da implantação programática ou virtual, uma informação existente StructureMetaData pré-determinada lê a informação de estrutura e a manuseia em um dos casos explicados no Manuseio da Informação StructuredMetaData.

Manuseio da Informação StructuredMetaData

Implantações baseadas no VFS
: O reconhecimento da estrutura é enviado para um conjunto de StructureDeployers.

Estruturas definidas de especificação JEE
: Nós temos implementações coincidentes de StructureDeployer

- EarStructure
- WarStructure

- JarStructure

DeclarativeStructures

Procure pelo arquivo **`META-INF/jboss-structure.xml`** dentro de sua implantação e analise-o para construir um **`StructureMetaData`** apropriado.

FileStructures

Apenas reconhece arquivos de configuração conhecidos, tais como arquivos **`-jboss-beans.xml`** ou **`-service.xml`**.

Exemplo 10.1. Uma amostra do `jboss-structure.xml`

```
<structure>
  <context
comparator="org.jboss.test.deployment.test.SomeDeploymentComparatorTop">
    <path name=""/>
    <metaDataPath>
      <path name="META-INF"/>
    </metaDataPath>
    <classpath>
      <path name="lib" suffixes=".jar"/>
    </classpath>
  </context>
</structure>
```

No caso do EarStructure, primeiro reconheça uma implantação de nível superior e depois processe recursivamente as sub-implantações.

Você pode implantar um **`StructureDeployer`** personalizado com a ajuda da classe **`GroupingStructure`** fornecida pela interface **`StructureDeployer`**.

Após ter reconhecido a estrutura de implantação, você pode passá-lo aos implantadores reais. O Objeto dos Implantadores sabe como lidar com os implantadores reais, usando um conjunto de implantadores por **`DeploymentStage`**.

Exemplo 10.2. Estágios de Implantação

```
public interface DeploymentStages {
   /** The not installed stage - nothing is done here */
   DeploymentStage NOT_INSTALLED = new DeploymentStage("Not Installed");

   /** The pre parse stage - where pre parsing stuff can be prepared; altDD,
ignore, ... */
   DeploymentStage PRE_PARSE = new DeploymentStage("PreParse", NOT_INSTALLED);

   /** The parse stage - where metadata is read */
   DeploymentStage PARSE = new DeploymentStage("Parse", PRE_PARSE);

   /** The post parse stage - where metadata can be fixed up */
   DeploymentStage POST_PARSE = new DeploymentStage("PostParse", PARSE);

   /** The pre describe stage - where default dependencies metadata can be
created */
   DeploymentStage PRE_DESCRIBE = new DeploymentStage("PreDescribe",
POST_PARSE);

   /** The describe stage - where dependencies are established */
   DeploymentStage DESCRIBE = new DeploymentStage("Describe", PRE_DESCRIBE);

   /** The classloader stage - where classloaders are created */
   DeploymentStage CLASSLOADER = new DeploymentStage("ClassLoader", DESCRIBE);

   /** The post classloader stage - e.g. aop */
   DeploymentStage POST_CLASSLOADER = new DeploymentStage("PostClassLoader",
CLASSLOADER);

   /** The pre real stage - where before real deployments are done */
   DeploymentStage PRE_REAL = new DeploymentStage("PreReal", POST_CLASSLOADER);

   /** The real stage - where real deployment processing is done */
   DeploymentStage REAL = new DeploymentStage("Real", PRE_REAL);

   /** The installed stage - could be used to provide valve in future? */
   DeploymentStage INSTALLED = new DeploymentStage("Installed", REAL);
}
```

Os estágios pré-existentes de implantações são mapeados aos estados de controlador interno do Microcontainer. Eles fornecem uma visualização cêntrica de ciclo de vida da implantação dos estados de controladores genéricos.

A implantação é convertida dentro do **`DeploymentControllerContext`** do componente do Microcontainer. A máquina do estado do Microcontainer manuseia as dependências.

As implantações são manuseadas sequencialmente pelo estágio de implantação. A ordem da hierarquia inteiramente implantada é manuseada usando a propriedade **`parent-first`** do implantador. Esta propriedade é determinada para **`true`** por padrão.

Você pode especificar também quais níveis de hierarquia seu implantador manusear. Você pode escolher **`all`**, **`top level`**, **`components only`** ou **`no components`**.\n\t\n

A maneira com que o Microcontainer manuseia os modelos de componente e manuseia dependências é válida aqui também. Caso haja uma dependência não resolvida, a implantação esperará no estado atual,

provavelmente relatando um erro caso o estado atual não seja um estado requerido.

A adição de uma nova implantação é concluída pela extensão de um dos muitos implantadores auxiliares existentes.

Alguns dos implantadores precisam da implantação de reforço VFS, enquanto outros usam uma implantação geral. Na maioria dos casos os implantadores de análise são os que precisam do reforço VFS.

Atenção

Além disso, certifique-se de que os implantadores rodam de forma recursiva através de toda a implantação, sub-implantação e componente. Seu código precisa determinar, o mais cedo possível no processo, se o implantador deve manusear a implantação ou não.

Exemplo 10.3. Implantador Simples que emite informação sobre a própria Implantação

```
public class StdioDeployer extends AbstractDeployer {
   public void deploy(DeploymentUnit unit) throws DeploymentException
   {
      System.out.println("Deploying unit: " + unit);
   }

   @Override
   public void undeploy(DeploymentUnit unit)
   {
      System.out.println("Undeploying unit: " + unit);
   }
}
```

Adicione esta descrição em um dos arquivos **`-jboss-beans.xml`** no diretório **`deployers/`** do Servidor do Aplicativo JBoss e o **`MainDeployerImpl`** bean selecionará este implantador através do manuseamento de retorno de chamada IoC do Microcontainer.

Exemplo 10.4. Descritor de Implantação Simples

```
<bean name="StdioDeployer" class="org.jboss.acme.StdioDeployer"/>
```

10.3. Controle de Fluxo Natural na forma de anexos

O VDF inclui um mecanismo chamado *attachments*, que facilita a passagem de informação de um implantador a outro. Os anexos são implantados como **`java.util.Map`** levemente aprimorado, sendo que cada entrada representa um anexo.

Alguns implantadores são produtores, enquanto outros são consumidores. O mesmo implantador pode executar também ambas funções. Alguns implantadores criam metadados ou instâncias de utilidade, colocando-as no mapa *attachments*. Outros implantadores apenas declaram sua necessidade para estes anexos e puxam os dados do mapa de anexos, antes de realizar o trabalho adicional naqueles dados.

O *Natural order* refere-se à maneira em que os implantadores são ordenados. Uma ordem comum e natural usa os termos relativos *before* e *after*. No entanto, com o mecanismo dos anexos já posicionados, você pode ordenar implantadores pela maneira em que eles produzem e/ou consomem os anexos.

Cada anexo possui uma chave e implantadores passam teclas aos anexos que produzem. Caso seu implantador produzir um anexo, a tecla produzida é chamada *output*. Caso o implantador consumir um anexo, a tecla é chamada *input*.

Os implantadores possuem entradas *ordinary* e entradas *required*. As entradas ordinary (simples) são apenas usadas para ajudar determinar a ordem natural. As entradas requeridas também ajudam a determinar a ordem além de outra função. Elas ajudam a determinar se o implantador é na realidade relevante para a implantação gerada, pela checagem se um anexo, que corresponde àquela entrada requerida, existe no mapa dos anexos.

Atenção

Enquanto a ordenação relativa continua sendo suportada, ela é considerada uma má prática e talvez não seja suportada nas liberações futuras.

10.4. Cliente, Usuário e Uso do Servidor e Detalhes da Implementação

Esses recursos ocultam os detalhes de implantação, reduzindo os erros de usagem, enquanto otimizando o processo de desenvolvimento.

O objetivo é que clientes apenas vejam um API de Desenvolvimento, enquanto desenvolvedores vejam um DeploymentUnit. Os detalhes da implementação do servidor podem ser encontradas no DeploymentContext. Apenas a informação necessária está exposta a um nível particular de ciclo de vida da implantação.

Os componentes já foram mencionados como parte do manuseio de hierarquia dos implantadores. Enquanto a implantação de nível superior e sub-implantadores é uma representação natural da hierarquia de estrutura da implantação, os componentes são um novo conceito VDF. A ideia dos componentes é que eles tenham um mapeamento 1:1 com o **`ControllerContexts`** dentro do Microcontainer. Consulte Por que Mapa de Componentes 1:1 com o **`ControllerContexts`** para maiores informações sobre este respeito.

Por que Mapa de Componentes 1:1 com o `ControllerContexts`

Nomeação
: O nome das unidades de componente do nome **`ControllerContext`**.

get*Scope() and get*MetaData()
: Retorna o mesmo contexto MDR que será usado pelo Microcontainer àquela instância.

IncompleteDeploymentException (IDE)
: Com o objetivo do IDE imprimir quais dependências faltam para a implantação, ele precisa saber os nomes do ControllerContext. Ele encontra o nome pela coleção dos nomes DeploymentUnit do Componente nos Implantadores do Componente que os especificam, tais

como **BeanMetaDataDeployer** ou o método **setUseUnitName()** no **AbstractRealDeployer**.

10.5. Máquina de Estado Único

Todos os componentes do Microcontainer são manuseados por cada ponto de entrada único ou máquina de estado único. As implantações não são exceção.

Você pode tirar vantagens deste recurso pelo uso do arquivo de configuração **jboss-dependency.xml** em suas implantações.

Exemplo 10.5. jboss-dependency.xml

```
<dependency xmlns="urn:jboss:dependency:1.0">
  <item whenRequired="Real" dependentState="Create">TransactionManager</item>
(1)
  <item>my-human-readable-deployment-alias</item> (2)
</dependency>
```

Note as chamadas artificiais no XML: (1) e (2).

(1) apresenta como descrever uma dependência em outro serviço. Esta amostra requer que o **TransactionManager** seja recriado antes da implantação estar no estado 'Real'.

(2) é um pouco mais complexo, uma vez que está faltando informação adicional. Por padrão, os nomes de implantação dentro do Microcontainer são nomes URI, que pela digitação dos mesmos os fazem uma proposição sujeita ao erro. Portanto, com o intuito de facilmente declarar uma dependência em outras implantações, você precisará de um mecanismo alias para evitar os nomes URI. Você pode adicionar um arquivo de texto plano nomeado **aliases.txt** em sua implantação. Cada linha do arquivo contém um alias, dando a um arquivo de implantação um ou mais nomes simples usados para referenciação.

10.6. Verificação de Classes para Anotações

As especificações JEE atuais reduzem o número de arquivos de configuração, mas o container é solicitado para realizar a maioria do trabalho usando @annotations. Com o objetivo de obter informação sobre o @annotation, os containers devem verificar as classes. Esta verificação cria uma desvantagem no desempenho.

Contanto, para reduzir a quantia de verificação, o Microcontainer fornece outro gancho descritor, por meio do **jboss-scanning.xml**.

Exemplo 10.6. jboss-scanning.xml

```
<scanning xmlns="urn:jboss:scanning:1.0">
  <path name="myejbs.jar">
    <include name="com.acme.foo"/>
    <exclude name="com.acme.foo.bar"/>
  </path>
  <path name="my.war/WEB-INF/classes">
    <include name="com.acme.foo"/>
  </path>
</scanning>
```

Estas amostras apresentam uma descrição simples dos caminhos relativos para inclusão ou exclusão quando verificando a Edição Java Enterprise de versão 5 ou maior anotada na informação de metadados.

Histórico de Revisão

Revisão 5-1.1.400	2013-10-31	Rüdiger Landmann

Rebuild with publican 4.0.0

Revisão 5-1.1	Fri Aug 31 2012	Leticia de Lima

Translation files synchronised with XML sources 5-1

Revisão 5-1	Wed Sep 15 2010	Misty Stanley-Jones

JBPAPP-5076 - Correção de desigualdades entre amostras e seus textos.
Número de versão alterada na linha com as novas solicitações de versão.
Revisado para a Plataforma do Aplicativo JBoss Enterprise 5.1.0.GA.